An Empirical Study of the Demeter System

Pengcheng Wu and Mitchell Wand

College of Computer and Information Science
Northeastern University
Boston, MA 02115, USA
{wupc,wand}@ccs.neu.edu

ABSTRACT

We studied a substantial application that used the Demeter toolset to determine how much manual coding was avoided by using the Demeter tools. Our study provides evidence that use of the Demeter tools avoids a considerable amount of manual coding.

1. INTRODUCTION

The Demeter system is a set of adaptive programming [8] tools to support better implementation of the task of traversing object structures in Java. It is a very common task in object-oriented programming to implement traversals on the object structures where a lot of low level and error-prone structural details have to be handled. Manual implementations of traversals are error-prone. Even a very careful programmer may write a traversal program traversing wrong paths or missing some paths in the object structure. Once those errors happen, debugging the traversal program is a difficult task, since the code implementing the traversals is usually tangled with the code for implementing other concerns.

Adaptive programming [8] supports high level descriptions (called *traversal strategies* or *strategy graphs*) of the traversal paths using graph primitives, which are automatically translated into the actual low level traversal code as implemented in DemeterJ [4] and DAJ [1] , or are interpreted at run time as implemented in DJ [10].

The goal of this work was to measure the extent to which use of the Demeter system freed programmers from tedious traversal code writing. To measure this, we took an algorithm [7] that simulated how a human programmer might generate traversal code and applied it to a substantial Demeter application, which is the DemeterJ compiler itself [4].

1.1 Traversal Generating Algorithms

The major difficulty in generating traversal code is that a traversal may pass through a class multiple times, and we need to keep track of where we are in the traversal. For example, consider the traversal *t1 = from A via B to E* for the class structure shown in Figure 1. A path for such a traversal would start at an A-Object. From there it must take the edge to a B-Object. After that, it might pass through any number of C-, D-, and A-Objects before reaching an E-Object. The traversal code must keep track where it is in the traversal graph, so that when the traversal reaches the second or later A-Object, it will know that it can then proceed to the E-Object.

The Demeter system uses the TGA algorithm [9] to generate code for traversals. This algorithm keeps track of its place in the traversal by passing an argument of type *BitSet* that records which classes have already been seen in the traversal. Thus we only need to generate at most one traversal method into each class for each traversal and the total number of the traversal methods needed for each traversal is linear in the size of the class graph in the worst case. Of course, the traversal methods generated by the TGA algorithm will have extra runtime overhead for checking the value of the *BitSet* argument.

The *FIRST* algorithm [7] uses a different technique. It generates traversal methods by calculating, for each pair of a source class and a set of target classes, the set of edges from the source class through which an object of one of the target classes is reachable. It uses these sets in conjunction with the strategy graph to generate the traversal methods. For the example *t1 = from A via B to E* above, the code shown in Listing 1 would be generated. Instead of passing a *BitSet* argument, the algorithm calls one of two methods in class A, depending on where it is in the strategy graph. This algorithm avoids the overhead of checking the *BitSet* argument, at the expense of adding potentially exponentially many new methods in the worst case [9] .

This code is similar to what a human programmer might produce. We need this form of traversal code instead of a field-access path like *this.b.c.d* because usually the objective of the traversal is to perform some process along the traversal, not just get the target objects. For example, we could attach *Visitor* objects [5] to the traversal methods to execute user-defined actions.

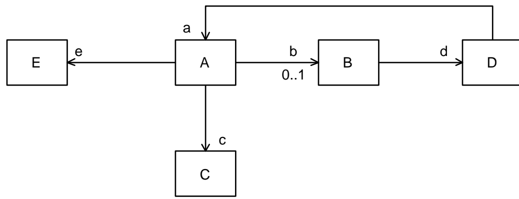

Figure 1: UML Diagram

Listing 1: Traversal methods generated by the FIRST set based algorithm

```
//for traversal: from A via B to E
class A {
  void t1() {
    if(b!=null) {
      b.t1();
    }
  }
  void t1prime() {
    access(e);
    if(b!=null) {
      b.t1();
    }
  }
}

class B {
  void t1() {
    d.t1();
  }
}

//class C is not involved in this traversal

class D {
  void t1() {
    a.t1prime();
  }
}
```

2. EXPERIMENTAL PROCEDURE

2.1 Test Bed

Although there have been a number of applications developed with the Demeter tools or methodology, including a mission-critical application developed at Verizon [2], the largest application we know of is the DemeterJ compiler itself [4].

To study how the use of traversals reduces the number of methods to be written manually, we studied the *generate* package of the DemeterJ's compiler, which is the largest package using traversal strategies in the implementation. It has 413 classes (including interfaces) and 80 traversals in total. All kinds of possible relationships between classes/interfaces are present in this package and for association relationships, all kinds of multiplicities are present. The package defines the syntax and the semantics of the DemeterJ's *class dictionary* language (for defining the class structure of a program) and its *traversal strategy* language (for specifying traversal paths over an object structure). The package also implements the TGA algorithm.

As in many other language processing systems, we need to write a large amount of traversal code over the Abstract Syntax Tree in the implementation of this package. The goal of this experiment was to study how hard it would have been to manually write those traversal code on such a complex class structure without tool support.

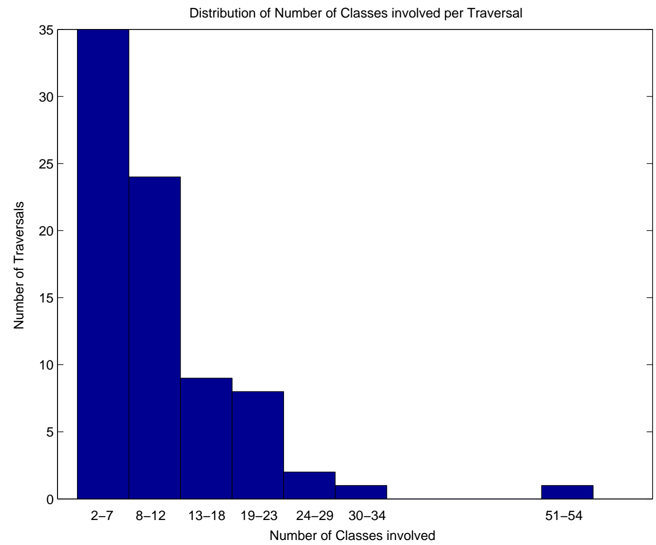

Figure 2: Traversals by Number of Classes Involved

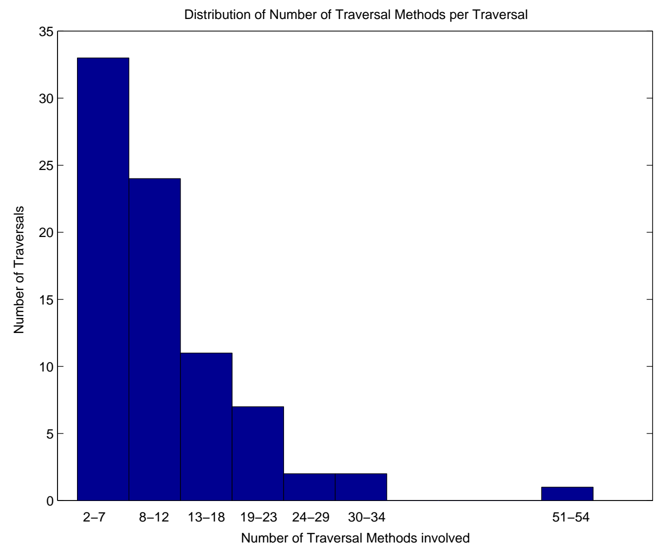

Figure 3: Traversals by Number of Methods Involved

To do this, we ran the *FIRST* algorithm on the 80 traversals and the class graph of the *generate* package and collected metrics on the generated methods. This algorithm closely mimics what a human programmer might generate for these traversals.

2.2 Metric 1: Number of classes/methods involved

To measure how much work have been saved for programmers by using Demeter's high level traversal specification and the FIRST set based traversal generating algorithm, the metric of how many traversal methods have been generated and how many classes have been involved would be a good indicator. For the 80 traversals in the *generate* package, the FIRST algorithm generated 871 methods involving 185 of the 413 classes in the class hierarchy.

To get the flavor of how complex the traversal programs could be for the 80 traversals, Figure 2. and Figure 3. are the histograms for the number of classes into which the traversal methods are

generated and for the number of traversal methods generated for each traversal. About 25% of the traversals are complex: the number of classes involved and the number of methods generated both are larger than 12. There is even a traversal whose number of classes involved and whose number of traversal methods generated are both larger than 50. Although the number is quite impressive, it turns out that the traversal specification itself is not that complex in that it only has the simplest form of *from ClassDef to* $\{$*PartName,ClassSpec*$\}$. That utility traversal strategy was written to reach two of the terminal nodes (*PartName* and *ClassSpec*) in a large class *ClassDef* so that a Visitor object may be attached to the traversal and perform corresponding process along the traversal. Another observation about the two histograms is that they almost have exactly the same shape. It suggests that a typical traversal in this package only needs one traversal method for each class involved. In this case, the FIRST algorithm is an ideal algorithm to be used for generating traversal methods. Figure 2. also shows the measure of scattering of the implementation of the traversal concern if we don't use the Demeter toolset or any other AOP languages or tools.

2.3 Metric 2: The abstractness of traversal strategies

One of the advantages that the Demeter tools can provide is that to implement a traversal that needs to go through a tediously long object access path, one doesn't have to specify the details of each intermediate node/edge along the path. Instead, one only needs to specify the path by giving *milestone* class nodes/edges along the path using intuitive graph primitives, including *via*, *bypassing* among others, then the traversal generating algorithms will handle with the remaining details. This implementation strategy makes the program less error-prone.

We use the measurement of *abstractness* of a traversal strategy to quantify the ignored details of the traversal. The more abstract a traversal strategy is, the more details are ignored and thus the program is more robust under structural changes.

First, we define the *length of a traversal strategy*. A traversal strategy is composed of a sequence of milestones, each of which is a set of class nodes or edges in the class diagram. Those milestones indicate the class nodes or edges the traversal has to go through or bypass. For example, traversal strategy *from* $\{A\}$ *via* $\{B,C\}$ *to* $\{D,E\}$ has three sequential milestones: {A},{B,C}, and {D,E}. The length of a traversal strategy is defined as

$$(\text{the number of milestones}) - 1$$

Then, we define the *length of traversal method call paths* for a traversal as the largest number of association relationship edges that a branch of the traversal methods call chain may cross, in which if there is a loop, each of the association relationship edges along the loop will be counted exactly once. The *abstractness of a traversal strategy* is defined as the ratio of the length of traversal methods call paths over the length of the traversal strategy, i.e.:

$$\frac{\text{The length of methods call paths}}{\text{The length of the traversal strategy.}}$$

The histogram (Figure 4.) of the abstractness of the traversal strategies in the *generate* package shows that quite a few traversal strategies in that package have a high abstractness. Almost one

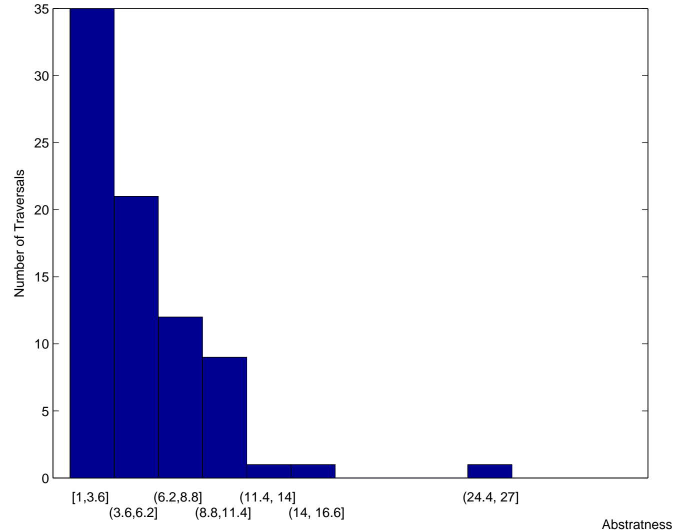

Figure 4: Traversals by Abstractness

third of the traversal strategies have the abstractness of larger than 6.2, in which 3 strategies even have the abstractness of larger than 11.4. Like the pattern we have found in Metric 1, the traversals having high abstractness usually are not complex traversals, instead, they tend to be simple straightforward traversals that try to reach some terminal nodes contained in a large object. For example, the simple strategy *from ClassDef to* $\{$*PartName,ClassSpec*$\}$ has the abstractness of 27.

2.4 Metric 3: The fan out of traversal strategies

Different traversal strategies may have different fan out properties. Some traversal strategies tend to be very "thin" in that at each class node along the traversal, there is only 1 association relationship edge branch to go down to reach the target nodes; while some strategies tend to have bigger fan out metrics in that at each class node along the traversal, there may be multiple association relationship edge branches to go down to reach the target nodes. We use formula

$$\frac{\text{Number of traversal methods}}{\text{Length of traversal methods call paths}}$$

to approximate the fan out of a traversal strategy. Figure 5. is the histogram of the fan out metrics of traversal strategies in the package. As you can tell from the figure, the traversals in this package tend to be quite "thin", i.e., usually there is only one association relationship edge from a class that can lead the traversal to reach the target nodes. The diagram is consistent with our another statistics of the FIRST set size for all the traversals in the package, where the average size of the FIRST sets of all potential pairs of classes is very close to 1. Thin traversal strategies indicate that manual implementation of the traversal methods would not be very difficult, instead, the implementation could be just very tedious, which supports one of our suspicions about the software engineering help the Demeter tools can offer.

2.5 Metric 4: The distribution of traversals over classes and the association relationships

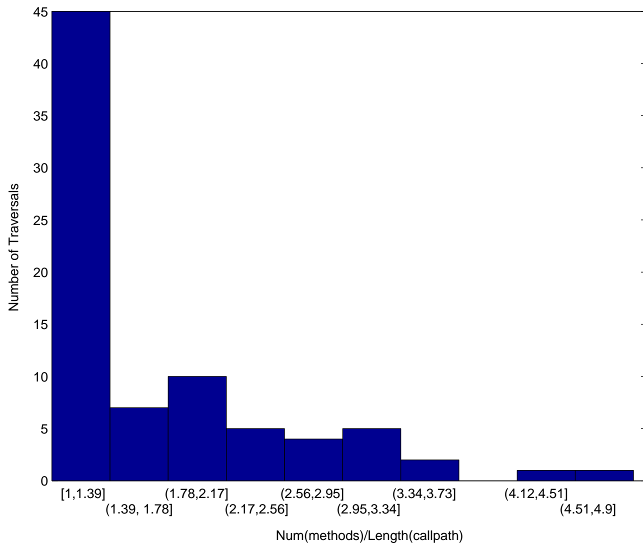

Figure 5: Traversals by Fan Out

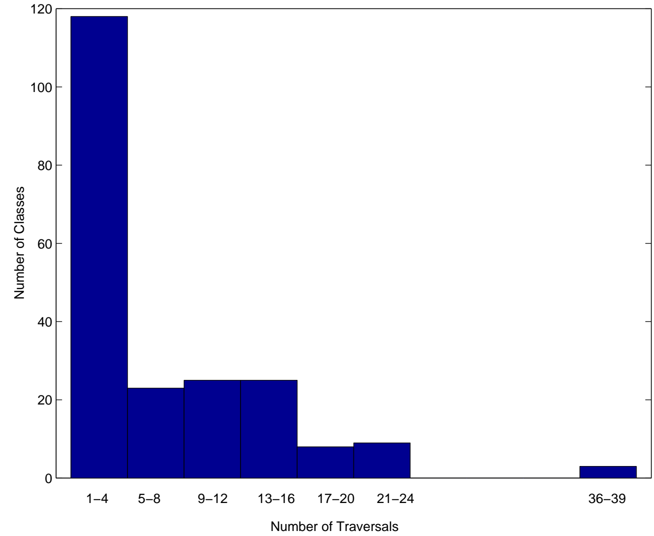

Figure 6: Classes by Traversals Involved

Usually in a class structure diagram, some classes are more "important" than other classes in the sense that the former are referred much more frequently than the latter. It is also the case for traversal strategies, i.e., some classes are referred much more frequently than the rest. We call those classes the *key* classes.

By observing what classes are the key classes in program, we can also determine which classes need special attention at design phase. To make a program easier to evolve, we need to deliberately design those key classes at the very beginning in such a way that we consider the possible requirement changes in the future and reflect those considerations in the key classes. Those special deliberation will make it unlikely that those key classes' structures need to be changed in the future when the predicted requirement changes really happen, thus the maintenance cost is reduced.

Figure 6. shows the histogram of the number of traversals a class in the package may be involved. Most of the classes are only involved in less than 5 traversals. However, we do have 3 classes involved in nearly 40 traversals. These 3 classes (class *Definition*, *ClassDef* and *ClassGraphEntry*) are the core framework classes defining the class graph structure of a program, thus a lot of traversals need to go through them to access the primary information.

Similar results are obtained for the histogram (Figure 7.) of the number of traversals an association relationship in the package may be involved. The result is consistent with the result of the distribution of traversals over classes, since the most frequently referred to association edges are the edges directly related to the key classes (This is not directly shown in the data collected. Instead, we manually checked the most frequently referred edges to see how many of them are directly related to the key classes.) .

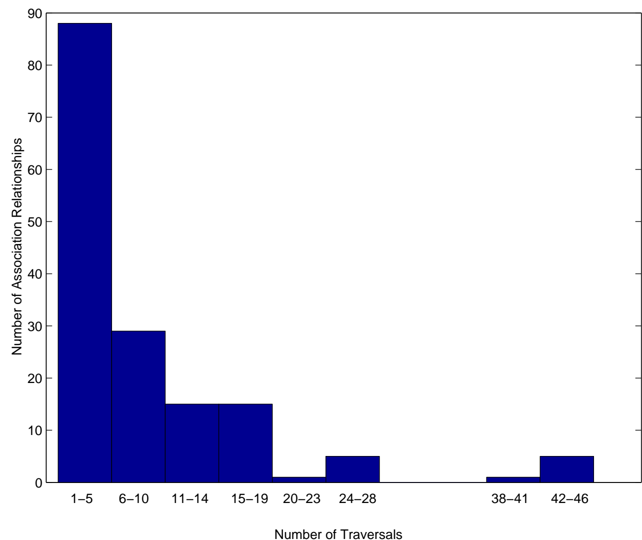

Figure 7: Edges by Traversals Involved

3. RELATED WORK

There has been some work in evaluating the software-engineering gains from the use of aspect-oriented techniques in real applications.

Coady and Kiczales [3] present their work on using AspectJ [6, 11] to refactor the FreeBSD operating system to analyze how As-

pectJ might help improve software evolution. They first introduce several aspects to the version 2 of the FreeBSD operating system, then they roll them forward into their subsequent incarnations in version 3 and 4 respectively. Their results show that the key benefits of the AOP implementation are localized changeability, explicit configurability, reduced redundancy, and subsequent modular extensibility.

Zhang and Jacobsen [12] give their empirical study on how AspectJ can be used to refactor the implementation of middleware platforms. They first investigated several middleware implementations and identified that they all have crosscutting concerns. Then they use AspectJ to refactor the implementations of those crosscutting concerns. Their conclusion is that AspectJ does improve the implementations of those concerns, while the runtime performance remains the same as before.

4. CONCLUSION

We conclude this paper by summarizing the key points in the format required by the workshop organizers.

4.1 Solution Name

Adaptive programming tools (Demeter system).

4.2 Problem Addressed

The problem that the Demeter system addresses is to separate the implementation of traversal related concerns from the implementation of other concerns.

4.3 Brief Description of Demeter System

The Demeter system supports a high level descriptive language called *traversal strategy* for programmers to specify the traversal paths in an object structure. The actual traversal will be embodied in either the traversal methods generated from the traversal strategies by traversal generating algorithms or the run time interpretation of the traversal. Visitor objects can be attached to a traversal for user-defined actions along the traversal.

4.4 Comprehensibility

Given that it is not unusual that a traversal may have a large number of classes and methods involved (Metric 1) and a traversal may have a very long traversal path while the traversal semantics could actually be very simple (Metric 2), high level descriptive traversal strategies capture the essences of traversals and thus promote the comprehensibility.

4.5 Semantic Interactions

Although not directly shown in any metric, the fact that the Demeter system uses the Visitor pattern for programmers to define customized processing along the traversals has already improved the semantic interactions between the implementation of the traversal concern and other concerns.

4.6 Summary

We did an empirical study on what software engineering benifits the Demeter system can provide in a real application to improve the implementation of traversal related concerns and show the benefits are nontrivial.

5. REFERENCES

[1] DAJ home page at sourceforge. http://daj.sourceforge.net. Continuously updated.

[2] Luis Blando. The eel compiler using adaptive object oriented programming and demeter/java. In *http://www.blando.info/luis/eelc/eelc_v20.htm*. Apr 1997.

[3] Yvonne Coady and Gregor Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59. ACM Press, 2003.

[4] Demeter Research Group. Online Material on Adaptive Programming and Demeter. In *http://www.ccs.neu.edu/research/demeter/*. Northeastern University, 1989-2003.

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

[6] 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*, pages 327–353, Budapest, 2001. Springer Verlag.

[7] Karl Lieberherr and Mitchell Wand. Traversal semantics in object graphs. Technical Report NU-CCS-2001-05, College of Computer Science, Northeastern University, May 2001.

[8] 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 http://www.ccs.neu.edu/research/demeter/biblio/dem-book.html.

[9] Karl J. Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of Object Structures: Specification and Efficient Implementation. *ACM Transactions on Programming Languages and Systems*, March 2004. To appear, currently available at: http://www.ccs.neu.edu/research/demeter/papers/strategies/revised-toplas/final/strategies.ps.

[10] 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.

[11] AspectJ Team. AspectJ home page. http://www.eclipse.org/aspectj. Continuously updated.

[12] Charles Zhang and Hans-Arno. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 130–139. ACM Press, 2003.