

# Investigating the Relationship Between Violations of the Law of Demeter and Software Maintainability

Jeffrey D. Palm  
University of Colorado  
430 UCB  
Boulder, CO, 80309-0430  
jdp@cs.colorado.edu

Kenneth M. Anderson  
University of Colorado  
430 UCB  
Boulder, CO, 80309-0430  
kena@cs.colorado.edu

Karl M. Lieberherr  
Northeastern University  
360 Avenue of the Arts  
Cullinane Hall, Boston, MA  
02115-5000  
lieberherr@ccs.neu.edu

## ABSTRACT

We present **DemeterCop** – a system for finding violations of the Law of Demeter in Java<sup>TM</sup> code while investigating the correlation between these violations and software maintainability. We compare how fixing these violations affects the Maintainability Index – hence the software maintainability – and look into new ways of fixing these violations.

## Categories and Subject Descriptors

D.1.0 [Programming Techniques]: Object-oriented Programming; D.2.3 [Software Engineering]: Coding Tools and Techniques— *Object-oriented programming*; D.2.8 [Software Engineering]: Metrics—*Process metrics*

## 1. INTRODUCTION

The Law of Demeter (LoD) is a programming-language independent rule using many aspect-oriented concepts that states that objects should only send messages to other *closely-related* objects [17]; we believe that violations of this rule adversely affect the maintainability of software. This work presents a system called **DemeterCop** [7] that identifies violations of the LoD in Java code in hopes of finding a connection between these violations and the maintainability of software. One of the largest costs in software development is maintenance, yet much of the current focus in object-oriented and aspect-oriented languages is placed on alternative factors. Hewlett-Packard estimates that 60% to 80% of its R&D personnel are involved in maintaining existing software, and that 40% to 60% of production costs are directly related to maintenance [6]. Of the many factors affecting the maintainability of software – Martin and McClure give seven [18] – we focus on the complexity of programs as indicators of their maintainability. In particular, we use the Maintainability Index (MI) [20], which combines a number of well-known measures, to quantify a software system’s maintainability and compare how violations of the LoD affect this index.

This paper introduces the LoD in section 2, the MI used in section 3, the **DemeterCop** system in section 4, preliminary results in section 5, related work in 6, and concludes with conclusions and future plans in section 7. The contribution of this work is an increased understanding of how violating the LoD can affect a software system’s maintainability and a solution that fixes these violations to improve a system’s MI.

## 2. LAW OF DEMETER

The Law of Demeter [15] is a rule applied to programming languages that improves the quality of object-oriented code [16] by restricting the coupling between classes; this coupling occurs at call sites when one object sends a message to another. The law, in general, states that an object can only send messages to *closely-related* objects, in one of two forms:

- In the *class form*, methods  $M$  of class  $C$  may only send messages to
  1. Instance variables of class  $C$ ,
  2. Argument classes to  $M$ ,
  3. Classes of objects created in  $M$ , or
  4. Classes of global variables used in  $M$ .
- In the *object form*, methods  $M$  of class  $C$  may only send messages to
  1. Immediate parts of  $C$ ,
  2. Argument objects to  $M$ ,
  3. Objects created in  $M$ , or
  4. Objects in global variables.

The main difference between the two is that the class form can be checked statically, whereas the object form cannot. For this reason our system uses the class form. There are some caveats to this law, though, and they are discussed in section 4.

The motivation behind LoD is to ensure that software is as modular as possible [15] and reduce the number of nested message sends. Such a violation is shown in Listing 1. This code violates the law because the receiver of the `nothing`

Cyc. Comp.	Risk
1-10	a simple program, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk program
greater than 50	untestable program (very high risk)

Table 1: Cyclomatic Complexity vs. Risk Evaluation

method is not an instance variable of  $C$ , an argument to `violation`, created in `violation`, or a global variable. This is a primitive example of a traversal that is seen at the site of a violation; a more drastic example would be something such as `o.a().b().c().d().e()`, where one clearly sees the programmer is navigating from the the object `o` to the message `e`. We have developed the `DemeterCop` system primarily to detect and fix this type of violation.

Listing 1: Simple LoD Violation

```

1 class A { void nothing() {} }
2 class B { A a; a() { return a; } }
3 class C {
4     void violation(B b) { b.a().nothing(); }
5         // Violation! ~~~~~
6     }
7 }

```

### 3. MAINTAINABILITY INDEX

The Maintainability Index, given in Equation 1, is a combination of the average cyclomatic complexity per module<sup>1</sup>  $aveV(g')$ , average halstead volume per module  $aveV$ , and average lines of code per module  $aveLOC$ , and is a measure of software maintainability where higher MIs denote higher maintainabilities.

$$MI = 171 - 5.2 \times \ln aveV - 0.23 + 50 \times aveV(g') - 16.2 \times \ln aveLOC \quad (1)$$

The cyclomatic complexity of a program measures the number of linearly-independent paths through a program module's control flow graph [19] and is given in Equation 2.

$$Cyclomatic\ Complexity = E - N + p \quad (2)$$

In this equation  $E$  is the number of edges of the graph,  $N$  is the number of nodes of the graph, and  $p$  is the number of connected components. Table 3 gives a rough idea of how to interpret a cyclomatic complexity number [1].

The Halstead Effort measures the complexity of a program straight from the source code's operators and operands [10] and is the product of the Halstead *Volume* and *Difficulty*.

$$Halstead\ Effort = \frac{n_1}{2} \times \frac{N_2}{n_2} \times (N_1 + N_2) \times \log_2 n_1 + n_2 \quad (3)$$

<sup>1</sup>A module was taken to be `.java` file

Measure	Value
$aveV$	2183.37
$aveV(g')$	138.76
$aveLOC$	87.03
MI	26.75

Table 2: MI for Jakarta Ant

In this equation  $n_1$  is the number of distinct operators,  $n_2$  is the number of distinct operands,  $N_1$  is the total number of operators,  $N_2$  is the total number of operands, the sum  $N_1 + N_2$  is said to be the program length, and the sum  $n_1 + n_2$  is said to be the program vocabulary.

Table 3 shows the MI breakdown for a typical Java project – Jakarta Ant[12].

## 4. DEMETERCOP SYSTEM

The `DemeterCop` system is implemented as a Java Development Tool (JDT) extension to the Eclipse™ development environment[2] and can be run in parallel with other language tools such as compilers, code navigators, and formatters. This system currently operates on a Java project and (1) finds all violations in the project, (2) finds the MI in the project, and (3) provides primitive solutions to violations. In this section we discuss the system's current operation, the algorithm for finding violations, and future plans.

### 4.1 Current Operation

`DemeterCop` is applied to a Java project using a context-sensitive menu on the Eclipse project tree. When invoked, `DemeterCop` performs its operations (as described below) and presents violations (if any) to the user in a number of ways. Currently, violations are indicated by highlighting files in the project's source tree (1), underlining violations within a source file (2), adding items to a developer's task list (3), and highlighting methods in the project's method tree (4). Figure 1 displays each of these techniques after `DemeterCop` found 300 violations in the `htmlparser` project [11].

### 4.2 The Algorithm

The algorithm for finding whether a message send violates the LoD involves two inspections; the first analyzes the actual receiver of the message and the second does a flow analysis to see if any previous assignments to that receiver could result in a violation.

The latter inspection is very important, since if a single path to the call site contains a violation, then the call site itself is in violation. For example, in Listing 2, class `C` calls the `toString` method of object `o` on line 8. However, class `C` must take one of two paths to make that call. The assignment on line 6 is fine because the object `o` is created within the current method. But, the assignment on line 7 creates a violation as described above. As such, the call on line 8 is a violation because there exists a path to that point where a violation may occur.

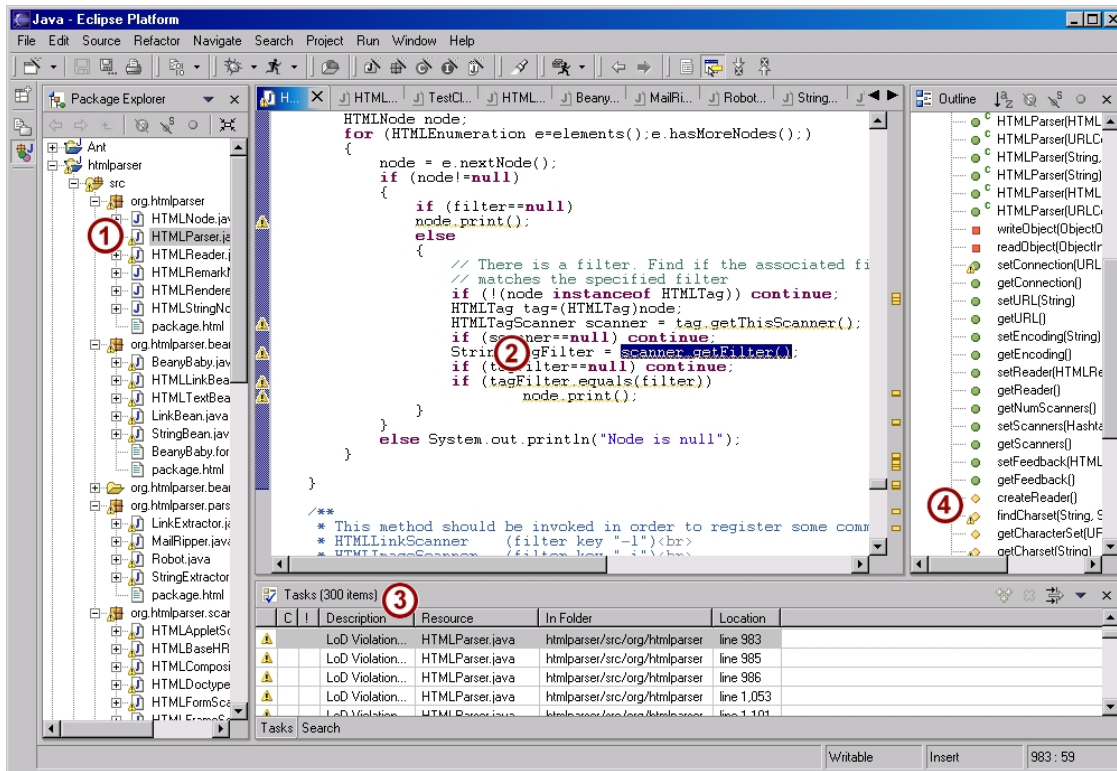


Figure 1: DemeterCop screen shot

Listing 2: A violation due to assignment

```

1 class A { void nothing() {} }
2 class B { A a; a() { return a; } }
3 class C {
4   void violation(B t, boolean b) {
5     Object o;
6     if (b) o="s"; // String->toString: OK
7     else o=t.a(); // B->A->toString: No
8     String s=o.toString(); // Violation
9   }
10 }

```

The algorithm to detect violations is as follows:

1. **inspectSimple** takes an AST node that is the receiver  $R$  of a call site and a **boolean** value to return on success and returns whether this call site is in violation of LoD. If any of the following is true of this receiver at the call site, we return the success value:
  - $R$  is an explicit **this** reference.
  - $R$  is a parameter to the enclosing method or constructor.
  - $R$  is a global object.
  - $R$  is a computed object – i.e. an object returned by an instance method.
  - $R$  is a stored object – i.e. a field of the enclosing object.
  - $R$  was created in the enclosing method, constructor, or initializer

- $R$  is an instance of a stable class. It is currently believed that instances of “stable” classes may be sent any message, and we classify all classes specified in the Java Platform 1.4.0 [13] to be stable.

2. **inspect** is the main entry point and takes an AST node that is the receiver  $R$  of a call site. This routine first calls **inspectSimple** with a success value of **true**, and if this returns **true**, the call site is marked as a violation. If **inspectSimple** returns **false** we then do a flow analysis by iteratively calling **inspectSimple** on every assignment leading to  $R$ 's call site.
3. **flowAnalysis** takes an AST node  $R$  and first builds a control flow graph for the method, constructor, or initializer enclosing  $R$ . It then iterates over every path leading to  $R$ 's call site and calls **inspectSimple** with a success value of **false**. This call to **inspectSimple** will return **true** if that call site is in violation and **false** if the site is a violation. Hence, we stop after finding the first **true**, because only one violating assignment needs to exist to make this call site a violation.

### 4.3 Fixing the Violations

As discussed in section 5, preliminary results indicate a connection between LoD violations and software maintainability. As such, we are now developing techniques to fix violations in an automated fashion. The simplest way to fix a call site that is sending a message to an unrelated object is to introduce a new method. For example if `a.c().b()` is a violation, we simply can add a method, say `cb(){return c().b();}` on `a`'s class. But this does not increase the MII, because, intuitively, it increases the lines of code and does not affect

the cyclomatic complexity or Halstead Complexity.

Another solution is to use existing tools such as DJ [8] – a Java library for expressing traversals – or AspectJ<sup>TM</sup> [14] to turn hard-coded traversals into a more adaptive form. For example, the code in Listing 3 shows a traversal travelling between (1) `HTMLEnumeration`, (2) `HTMLTag`, (3) `HTMLTagScanner`, and (4) `String`.

Listing 3: A violation in `HTMLParser` code

```
1 public void parse(String filter) {
2   HTMLNode node;
3   for (HTMLEnumeration e=elements(); /*1*/
4     e.hasMoreNodes();) {
5     HTMLTag tag=
6       (HTMLTag)e.nextNode(); /*2*/
7     HTMLTagScanner scanner =
8       tag.getThisScanner(); /*3*/
9     if (scanner==null) continue;
10    String tagFilter =
11      scanner.getFilter(); /*4*/
12    if (tagFilter==null) continue;
13    if (tagFilter.equals(filter))
14      tag.print();
15  }
16 }
17 }
```

## 5. PRELIMINARY RESULTS

We have applied `DemeterCop` to a variety of open source Java projects and are collecting data to determine if there is a correlation between large numbers of LoD violations and poor MI ratings. While our results are preliminary, our initial data seems to indicate that this correlation exists. We are now analyzing the data in depth to understand the strength of this correlation.

## 6. RELATED WORK

Basili *et al.* studied various object-oriented design metrics as quality metric indicators, and, in particular, considered the coupling between object classes (CBO) [3]. According to this work, a class *A* is coupled to another class *B* if *A* uses *B*'s member functions or instance variables, and the CBO is the number of classes to which a class is coupled. Basili *et al.* predict that highly-coupled classes are more fault-prone than weakly-coupled classes. Since a higher CBO indicates a greater chance for error-prone classes, the LoD reduces the couple between classes, and error-prone classes are less maintainable, this suggests that code obeying the LoD will be more maintainable.

Briand *et al.* used the M-System, a measurement tool based on GEN++, and a previous suite of metrics [4] to analyze the C++ code of students from the University of Maryland Computer Science Department and, too, found that coupling makes classes more prone to error [5]. This work differs from Basili's in that they focus on coupling and cohesion and analyze various forms of the two, such as *import coupling*, *export coupling*, and *method overriding*; whereas Basili *et al.* only considered coupling between object classes. Their findings show that many different forms of coupling measures seem to capture similar dimensions in the data. Which is

to suggest that the coupling reduced by obeying the LoD would, too, make classes less error-prone, and, thus, more maintainable.

Lastly, Glasberg *et al.* claim that two best metrics for predicting fault-proneness are inheritance depth and export coupling. They found export coupling to be a much better indicator [9]. This, too, suggests such a coupling would contribute to poor maintainability.

## 7. CONCLUSIONS AND FUTURE WORK

We have presented the `DemeterCop` tool and shown how it has been used to find violations of the Law of Demeter and the Maintainability Index of Java programs in hopes of finding a correspondence between the two. There are currently two ways of fixing violations found and one shows promise of raising the MI of programs. Further work is needed to develop a more concrete method for fixing violations – whether it can be done with one method or a combination of various different methods. Also, we will develop a technique for identifying and extracting traversals from a collection of violations.

Our current focus is on developing new techniques to fix violations. In addition, we are looking for other ways, besides traversals, in which the LoD can be violated. Finally, we intend to continue analyzing ways in which LoD violations impact software engineering concerns, such as system understandability and performance.

## 8. ACKNOWLEDGMENTS

Thanks to Ken Anderson, Karl Lieberherr, and Paul Freeman for their feedback, and to Sergei Kojarski for his thoughts on stable classes.

## 9. REFERENCES

- [1] Carnegie mellon software engineering institute cyclomatic complexity web page, January 2003. <http://www.sei.cmu.edu/activities/str/descriptions/cyclomatic.html>.
- [2] Eclipse home page, January 2003. <http://eclipse.org/>.
- [3] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering*, 22(10):751–761, 1996.
- [4] L. C. Briand, P. T. Devanbu, and W. L. Melo. An investigation into coupling measures for c++. In *International Conference on Software Engineering*, pages 412–421, 1997.
- [5] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *The Journal of Systems and Software*, 51(3):245–273, 2000.
- [6] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *IEEE Computer*, 27(8):44–49.
- [7] Demetercop website, January 2003. <http://demetercop.sourceforge.net/>.

- [8] Dj website, January 2003.  
<http://www.ccs.neu.edu/research/demeter/DJ/>.
- [9] D. Glasberg, K. Emam, W. Melo, and N. Madhavji. Validating object-oriented design metrics on a commercial java application, 2000.
- [10] M. H. Halstead. *Elements of Software Science, Operating, and Programming Systems Series*, y, 1977.
- [11] Htmlparser project website, January 2003.  
<http://htmlparser.sourceforge.net/>.
- [12] Jakarta ant project website, January 2003.  
<http://jakarta.apache.org/ant>.
- [13] Java™2 platform, standard edition, v 1.4.0 api specification, January 2003.  
<http://java.sun.com/j2se/1.4/docs/api/>.
- [14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
- [15] K. J. Lieberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.
- [16] K. J. Lieberherr and I. Holland. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices*, 24(3):67–78, March 1989.
- [17] K. J. Lieberherr, I. Holland, and A. J. Riel. Object-oriented programming: An objective sense of style. In *Object-Oriented Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, number 11, pages 323–334, San Diego, CA, September 1988. A short version of this paper appears in *IEEE Computer Magazine*, June 1988, Open Channel section, pages 78-79.
- [18] J. Martin and C. McClure. *Software Maintenance: The Problem and its Solution*. Prentice-Hall, 1983.
- [19] T. J. McCabe and A. H. Watson. Software complexity. *Crosstalk, Journal of Defense Software Engineering*, 7(12):5–9, December 1994.
- [20] K. D. Welker and P. W. Oman. Software maintainability metrics models in practice. *Crosstalk, Journal of Defense Software Engineering*, 8(11):19–23, November/December 1995.