Object-Oriented Programming: An Objective Sense of Style

K. Lieberherr, I. Holland, A. Riel

161 Cullinane Hall, College of Computer Science
Northeastern University, 360 Huntington Ave., Boston MA 02115

Abstract

We introduce a simple, programming language independent rule (known in-house as the Law of Demeter^{TM}) which encodes the ideas of encapsulation and modularity in an easy to follow form for the object-oriented programmer. You tend to get the following related benefits when you follow the Law of Demeter while minimizing simultaneously code duplication, the number of method arguments and the number of methods per class: Easier software maintenance, less coupling between your methods, better information hiding, narrower interfaces, methods which are easier to reuse, and easier correctness proofs using structural induction. We discuss two important interpretations of the Law (strong and weak) and we prove that any object-oriented program can be transformed to satisfy the Law. We express the Law in several languages which support object-oriented programming, including Flavors, Smalltalk-80, CLOS, C++ and Eiffel.

Keywords: Object-oriented programming, programming style, design style, software engineering principles, software maintenance and reusability.

A short version of this paper appeared in IEEE Computer, June 1988, Open Channel, page 79.

1 Introduction

For the past two years we have been using object-oriented programming techniques in our research and in our teaching at the undergraduate and graduate levels. During this time we have asked ourselves many stylistic questions such as,

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1988 ACM 0-89791-284-5/88/0009/0323 $1.50

"When is an object-oriented program written in good style?", "Is there some formula or rule which one can follow in order to write good object-oriented programs?", "What metrics can we apply to an object-oriented program to determine if it is 'good'?", and "What are the characteristics of good object-oriented programs?".

In this paper we put forward a simple rule (now known in-house as the *Law of Demeter* or the *Law of Good Style*) which, we believe, answers these questions and helps to formalize the existing ideas that can be found in the literature [7] [17] [3]. We claim that our Law promotes maintainability and comprehensibility. To prove this in absolute terms would require a large experiment with a statistical evaluation. As the field of object-oriented programming is new, large object-oriented software developments which are able to provide such data are rare. Indeed, we hope to provide a guiding principle to help such developments. We have examined our own code (about fourteen thousand lines of Lisp/Flavors) and are convinced of the Law's benefits. The close relationships and implications between the Law and software engineering principles such as coupling control, information hiding and narrow interfaces are pointed out below. These provide the support for our claim.

The Law of Demeter is named after the Demeter system which provides a high-level interface to class-based object-oriented systems. The most novel aspect of the Demeter system is that we view a (parameterized) class definition as a (parameterized) language definition. This point of view allows us to provide a large number of useful utilities (written in a specific object-oriented language) which impressively simplify the programming task. Examples of generated utilities are: application development plans, application skeletons, parsers, pretty printers, type checkers, object editors, LL(1) corrections [9] [11]. The explanations and examples presented in this paper are writ-

ten in the notation of Demeter, a modified EBNF notation which is described later in this paper.

One of the primary goals of the Demeter system is to develop an environment which facilitates the evolution of a class hierarchy. Such an environment must provide tools for the easy updating of existing software (i.e. the methods (operations) which are defined on the class hierarchy). We are striving to produce an environment which will allow software to be 'grown' in a continuous fashion rather than in sporadic jumps, which undoubtably leads to major rewrites. This will lead to the fast prototyping and updating system development cycle which is common in the Artificial Intelligence community.

In order to achieve this flexibility we would like the programs being written to be 'well behaved' or 'well formed' in some sense. In other words we would like the programs to follow a certain style which enables them to be modified easily. This ease of modification is one criterion which characterizes a good object-oriented programming style.

This issue is not only relevant to the Demeter world, but is one which should be adopted by all programmers which use object-oriented programming techniques. In addition, every object-oriented programmer should know what is considered 'good object-oriented programming style' in order to write easily maintainable systems just as procedural programmers are aware of the top-down programming paradigm, the "thou shalt not use a goto" rule and others [8]. The Law can be understood without knowledge of Demeter, but the best formulation which is compile-time enforceable for a large class of object-oriented programs requires basic Demeter concepts.

In addition, we believe that programs which obey the Law will be more amenable to program verification (along the lines of [6]) and parameterization by example techniques. By the latter, we mean techniques which will allow us to take an existing class hierarchy and associated software and produce a more general system by treating some classes as parameters.

We challenge object-oriented programmers to check whether their programs follow our Law and where they do not, to consider whether they should. We welcome any comments and/or examples which require a contradiction to the Law of Demeter.

The examples in this paper are written in the notation of Demeter, therefore section two is dedicated to a description of this notation. The sections which follow will define the Law of Demeter both formally and through examples, examining both practical and theoretical issues. We present a proof which states that any object-oriented program[1] written in bad style can be transformed systematically into a structured program obeying the Law of Demeter. The implication of this proof is that the Law of Demeter does not restrict what a programmer can solve, it only restricts the way he or she solves it.

2 Notation

The class hierarchy in Demeter is described using three kinds of production rules. A collection of these production rules is called a class dictionary.

1. A *construction* production is used to build a class from a number of other classes and is of the form C=<id_1> SC_1 ... <id_n> SC_n. Here C is defined as being made up of n parts (called its instance variable values), each part has an identifier id_i (called an instance variable name) and a type SC_i (called an instance variable type). This means that for any instance (or member) of C the identifier id_i refers to a member of class SC_i. We use class and type as synonyms. The following example describes a library class as consisting of a reference section, a loan section, and a journal section.

   ```
   Library =
     <reference> Reference-Sec
     <loan> Loan-Sec
     <journal> Journal-Sec.
   ```

   The following naming convention is used : Instance variable names will begin with a lower case letter and class names will begin with a capital letter.

2. An *alternation* production allows us to express a union type. A production of the form $C : A|B.$ states that a member of C is a member of class A or class B (exclusively). For example,

   ```
   Book-Identifier :
     ISBN | Library-of-Congress.
   ```

   which expresses the notion that when we refer to the identifier of a book we are actually referring to its ISBN code or its Library of Congress code.

3. A *repetition* production is simply a variation of the construction production where all the instance variables have the same type and we do

[1] The proof is phrased in Lisp/Flavors notation but can be generalized to cover other syntax.

not specify the number of instance variables involved. The production $C \sim \{A\}$ defines members of C to be lists of zero or more instances of A.

We partially define a reference section in the library class in the following class dictionary.

```
Reference-Books-Sec =
  <ref-books> List-of-Books
  <ref-catalog> Catalog.
List-of-Books ~ {Book}.
Catalog ~ {Catalog-Entry}.
Book =
  <title> String
  <author> String
  <id> Book-identifier.
```

To express inheritance we use the notation *inherit* with a construction production. For example, every mathematics text book is a member of the class Book (as defined above), but it has some additional characteristics. We can express this by

```
Math-Text =
  <math-cat> Math-Category
    *inherit* Book.
Math-Category :
  Algebra | Calculus | Statistics.
```

An instance of Math-Text is also a member of Book and will therefore contain the instance variables of both.

Currently the Demeter system "sits on top of" Old-Flavors and so the methods are written in the Lisp/Old-Flavors notation.

Before we introduce the Law, two definitions are appropriate:

- We call the set formed by taking the union of the set of methods attached to a class C and the set of methods attached to C's super classes, the *signature* of C.
- We define a set of classes associated with a given class, called *associated(C)*. If C is defined by an alternation production (e.g. $C : A|B.$) then the set *associated(C)* is the union of the classes associated with the alternatives in the right-hand-side of C's production. If C is defined by a construction or repetition production *associated*(C) is C itself.

3 The Law of Demeter

For all classes C, and for all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes:

1. **The argument classes of M (including C).**
2. **The instance variable classes of C.**

(Objects created by M, or by functions or methods which M calls, and objects in global variables are considered as arguments of M.)

This Law has two primary purposes:

1. Simplifies modifications. It simplifies the updating of a program when the class dictionary is changed.
2. Simplifies complexity of programming. It restricts the number of types the programmer has to be aware of when writing a method.

The Law of Demeter, when used in coordination with three key constraints, enforces good programming style. These constraints require minimizing code duplication, minimizing the number of arguments passed to methods and minimizing the number of methods per class.

4 The Motivation and Explanation

The motivation behind this Law is to ensure that the software is as modular as possible. Any method written to obey this Law will only know about the immediate structure of the class to which it is attached. The structure of the arguments and the sub-structure of C are hidden from M. Therefore, should a change to the structure of the class C be necessary we need only to look at those methods attached to C and its subclasses for possible conflicts. The Law effectively reduces the occurrences of certain nested message sendings (generic function calls) and simplifies the methods. The Law *prohibits* the nesting of generic accessor function calls, which return objects that are not instance variable objects. It *allows* the nesting of constructor function calls. An *accessor* function is a function which returns an object which did exist before the function is called. A *constructor* function returns an object which did not exist before the function is called.

The Law of Demeter has many implications regarding widely known software engineering principles. Our contribution is to condense many of the proven principles of software design into a single statement

which can easily be used by the object-oriented programmer and which can be easily checked at compile-time.

Some of the inter-related principles covered by the Law are the following:

- Coupling control.

  It is a well-known principle of software design to have minimal coupling between abstractions (e.g. procedures, modules, methods) [3]. The coupling can be along several links. An important link for methods is the "uses" link (or call/return link) which is established when one method calls another method. The Law of Demeter effectively reduces the methods we can call inside a given a method and therefore limits the coupling of methods with respect to the "uses" relation. The Law therefore facilitates reusability of methods and the abstraction level of the software.

- Information hiding.

  The Law of Demeter enforces one kind of information hiding: structure hiding. In general, the Law prevents a method from directly retrieving a subpart of an object which lies deep in that object's "part-of" hierarchy. Instead, intermediate methods must be used to traverse the "part-of" hierarchy in controlled small steps [12], [3].

  In some object-oriented systems, the user can protect some of the instance variables or methods of a class from outside access by making them private. This important feature complements the Law to increase modularity but is orthogonal to it. Our Law promotes the idea that the instance variables and methods which are public should be used in a restricted way.

- Information restriction.

  Our work is related to the work by Parnas et al. [15] [14] on the modular structure of complex systems. To reduce the cost of software changes in their operational flight program for the A-7E aircraft, the use of modules that provide information that is subject to change is restricted. We take this point of view seriously in our object-oriented programming and assume that any class could change. Therefore we restrict the use of message sendings (generic function calls) by the Law of Demeter. Information restriction complements information hiding. Instead of hiding certain methods, we make them public but we restrict their use.

- Localization of information.[2].

  The importance of localizing information is stressed in many software engineering texts. The Law of Demeter focusses on localizing type information When we study a method we only have to be aware of types which are very closely related to the class to which the method is attached. We can effectively be ignorant (and independent) of the rest of the system and as the old proverb goes: "Ignorance is bliss". This is an important aspect which helps to reduce the complexity of programming.

  From another point of view related to localization, the Law controls the visibility of message names. In a given method we can only use message names which are in the signatures of the instance variable types and argument types.

- Narrow interfaces.

  The maintenance of narrow interfaces between interacting entities is also important (see e.g. [12, page 303]). A method should have access to only as much information as it needs to do its job. If a method gets too much information, it has to destructure this information (via many nested sends) which the Law of Demeter discourages. Therefore The Law promotes narrow interfaces between methods.

- Structural induction.

  The Law of Demeter is related to the fundamental thesis of Denotational Semantics. That is, "The meaning of a phrase is a function of the meanings of its immediate constituents". This goes back to Frege's work on the principle of compositionality in his Begriffsschrift [5]. The main motivation behind the compositionality principle is that it facilitates structural induction proofs.

The Law is stated in terms of types and, as the following pathological example shows, its formulation does allow situations which violate the the principles that it is to enforce. Consider the class dictionary

```
A = <first> B <second> C
  <third> D <fourth> E.
B = <fifth> C <sixth> D.
D = <seventh> E.
```

and the method

```
(defmethod (A :bad-style)()
  (send (send (send-self :first)
    :sixth) :seventh))
```

[2] Peter Wegner pointed out this aspect of the Law.

All of the types in the body of this method, B, D, and E, are valid message receivers according to the Law. Yet the method looks two levels deep into the structure of instance variable first, violating the ideals of information-hiding and maintainability.

This problem can be removed by formulating the Law in terms of objects.

For all classes C, and for all methods M attached to C, all objects to which M sends a message must be

- **M's argument objects, including the self object or**
- **The instance variable objects of C.**

(Objects created by M, or by functions or methods which M calls, and objects in global variables are considered as arguments of M.)

By an argument object we mean an object passed by an argument. By an instance variable object we mean the object stored in an instance variable. From a conceptual point of view this seems the most natural way to state the Law. However, checking this Law at compile-time is complicated since it would involve a detailed analysis of aliases.

Consider

```
;A = <x> B.
(defmethod (A :alias) (t)
  (send self :set-x (send t :x)))

;T = <x> B.
(defmethod (T :m2) (a)
  (send (send a :x) :m3))
```

Is this in good style? It depends on the context. The following is o.k.

```
(send iA :alias iT) (send iT :m2 iA)
```

But `(send iT :m2 iA)` by itself violates the Law.

So, to retain easy compile-time checking we require the Law's formulation in terms of types. We feel that such pathalogical cases as the one above will not occur often enough to cause problems.

5 Example

We expand the Library definition in the above example into a nearly complete class dictionary.

```
Library =
  <reference> Reference-Sec
  <loan> Loan-Sec
  <journal> Journal-Sec.
Reference-Sec =
  <ref-book-sec> Books-Sec
  <archive> Archive.
Archive =
  <arch-microfiche> Microfiche-Files
  <arch-docs> Documents.
Microfiche-Files =
  <micro-list> List-of-Microfiche
  <micro-cat> Microfiche-Catalog.
Books-Sec =
  <books> List-of-Books
  <book-catalog> Catalog.
List-of-Books ~ {Book}.
Catalog ~ {Catalog-Entry}.
Book =
  <title> String <author> String
  <id> Book-identifier.
Book-Identifier : ISBN | Library-of-Congress.
```

Suppose we wish to attach a method to the class Library which will search its reference section for a specific book.

```
(defmethod (Library :Ref-Search)
    (book :type Book-Identifier)
  (send reference :Ref-search book))
```

The class Library simply passes the message on to class Reference-Sec.

```
(defmethod (Reference-Sec :Ref-search)
    (book :type Book-Identifier)
  (or (send ref-book-sec :search book)
(**)  (send
        (send archive :arch-microfiche)
          :search book)
(**)  (send
        (send archive :arch-docs)
          :search book)))


(defmethod (Microfiche-Files :search)
  (book :type Book-Identifier)
        ..... )
(defmethod (Documents  :search)
  (book :type Book-Identifier)
        ..... )
(defmethod (Books-Sec  :search)
  (book :type Book-Identifier)
        ..... )
```

The Ref-search method attached to Reference-Sec passes the message on to its book, microfiche and

document sections as explained below. This method breaks the Law of Demeter. The first message marked (**) sends the message "arch-microfiche" to "archive" which returns an object of type "Microfiche-Files". The method next sends this returned object the "search" message. However, "Microfiche-Files" is not an instance variable or argument type of class "Reference-Sec". Since the structure of all the classes are clearly defined by the class dictionary, we might be tempted to accept the above methods as a reasonable solution. Let us consider a change to the class dictionary. Assume the library installs new technology and replaces the microfiche and document sections of the archive with CD-Roms or Video-Discs.

```
Archive = <cd-rom-arch> CD-Rom-File.
CD-Rom-File =
  <cd-c-system> Computer-system
  <discs> CD-Rom-Discs.
```

We now have to search all of the methods, including the Ref-search method, for references to an archive with microfiche files. It would be easier to contain the modifications only to those methods which are attached to class Archive. We accomplish this by rewriting the methods in good style.

```
(defmethod (Reference-Sec :Ref-Search)
    (book :type Book-Identifier)
  (or (send ref-book-sec :search book)
      (send archive :search book)))

(defmethod (Archive :search)
    (book :type Book-Identifier)
  (or (send arch-microfiche :search)
      (send arch-docs :search)))
```

Notice how the coupling with respect to the "uses" relation has been reduced. Reference-Sec was coupled with Books-Sec, Archive, Microfiche-Files and Documents in the original version. Now it is coupled only with Books-Sec and Archive.

For a discussion of the complexity of programming point, consider the following example:

```
(defmethod (C :M) ()
  (send (send a :m1) :m2))
```

where a is an instance variable of C and m1 and m2 are user-defined methods (not instance variable access methods). Let's assume that m1 does not return an object which is of an instance variable type of C. Therefore the nested send expression is in bad style. But this expression does not make it easier nor harder to modify the methods when the class dictionary changes. However, it requires the programmer to think about other types than the instance variable types of C. The above method can be rewritten in good style as

```
(defmethod (C :M) ()
  (send self :m3 (send a :m1)))

(defmethod (C :m3) (arg :type Argtype)
  (send arg :m2))
```

Here the additional type is made explicit as an argument type. Sometimes it is possible to rewrite a program of the above form without introducing additional arguments.

6 The Trade-off

Writing programs which follow the the Law of Demeter decreases the occurrences of nested message sending and decreases the complexity of the methods, but it increases the number of methods. The latter is related to the problem outlined in [12] which is that there can be too many operations in a type. In this case the abstraction may be less comprehensible, and implementation and maintenance are more difficult. There might also be an increase in the number of arguments passed to some methods.

One way of correcting this problem is to organize all the methods associated with a particular functional (or algorithmic) task into "Modula-2 like" module structures as outlined in [11]. So the functional abstraction is no longer a method but a module which will hide the lower-level methods which caused the original confusion.

7 The Interface

Suppose we want to send a message to a reference section which is to return its book catalog. The method definition might look like the following.

```
(defmethod (Reference-Sec :return-book-cat) ()
  (send ref-book-sec :book-catalog))
```

The above method appears to obey the Law of Demeter, since we are sending an instance variable type of Reference-Sec a message. However, on closer inspection we see that the message being sent is the name of an instance variable which is not a part of Reference-Sec. This creates a situation in which the above method is sensitive to changes within the structure of the Books-Sec class. This clearly violates the spirit of the Law. The solution to this problem is to rewrite the methods as follows.

```
(defmethod (Reference-Sec :return-book-cat) ()
  (send ref-book-sec :return-book-cat))

* (defmethod (Books-Sec :return-book-cat) ()
    (send-self :book-catalog))
```

The method marked with an asterisk at first sight seems to be just a renaming of the instance variable name at the cost of one more method look-up. This is true but is better explained as the introduction of an interface between the Reference-Sec object and the Books-Sec object. This kind of interface has some very real advantages when it comes around to future updating of the software [17]. CLOS allows automatic generation of such an interface. A full interface of this sort would provide methods for accessing and setting the instance variables and thus hiding all the implementation details. Should the class be modified at a later stage these interface methods may be changed in an upward compatible way. The approach can also be taken when creating objects.

The Flavors mechanism for creating new objects (make-instance) uses the class instance variable names as keyword parameters in the make-instance call. This implies that a change in the structure of a class requires a search through the software for the make-instance calls for this class. One way of avoiding this is to use special "factory objects" (similar to those found in Objective-C [2]). For each application class, the interface has an associated "factory class" with one instance and one method called "make". This method contains the only make-instance call for the application class. The programmer can then change the "make" methods should a change in the class-dictionary be necessary.

8 The Weak and Strong Law of Demeter

The Law of Demeter shares many characteristics with other man-made laws. One such characteristic is that the Law is ambiguous and therefore open to interpretation. The source of the ambiguity is the rule which states that messages may only be sent to objects which are instances of classes associated with instance variable types of the class to which the method is attached. When we use the term instance variables do we mean the instance variables which make up the class exclusively, or are inherited instance variables also allowed. The question divides those who follow the Law of Demeter into two fundamental groups. The first group follows the Weak Law of Demeter while the second group adheres to the Strong Law of Demeter. The two versions of the Law are defined as follows.

- The Strong Law of Demeter: The Strong Law of Demeter defines instance variables as being ONLY the instance variables which make up a given class. Inherited instance variable types may not be passed messages.
- The Weak Law of Demeter: The Weak Law of Demeter defines instance variables as being BOTH the instance variables which make up a given class AND any instance variables inherited from other classes.

Each version carries certain implications. When the Strong Law of Demeter is adopted then it is guaranteed that any change to the underlying data structure will only affect methods attached to the changed classes. All methods which are attached to unaltered classes will not require modification. This allows the user to easily detect code which may require updating due to changes in the class hierarchy. Similarly, users which adopt the Weak Law of Demeter will gain certain advantages in program maintenance. However, these advantages are not as powerful as those gained in adopting the Strong Law. In the event of a change to the underlying data structure, the methods which may need modification are those attached to the altered classes OR any class which is inherited by an altered class. While the Strong Law appears to have a great advantage we will see that this advantage is not entirely free. The code which is written under the Strong Law tends to have extra methods for a given solution. This can render the code less readable in some cases.

The following example illustrates the issues through a sample solution to the problem of weighing a basket of Fruit. In this problem a basket of fruit is defined as a basket and a collection of various fruits. Each fruit has an attribute called weight which is assumed to be a number representing the weight. For simplicity we assume the basket doesn't have a weight. The underlying data structure is shown in the following class dictionary.

```
FruitBasket = Basket Fruits.
Fruits ~ { Fruit }.
Fruit : Apple | Orange | Plum
  *common* <weight> Number.
Basket = . Apple = . Orange = . Plum   = .
```

It is important to note that the classes Apple, Orange, and Plum all inherit from Fruit, and as a consequence also inherit the instance variable weight. We

assume that the weight for each piece of fruit has been stored in this inherited instance variable (possibly by some other method). We now want to write a method which will compute the prepared weight of a basket of fruit. The prepared weight of a piece of fruit is the weight of the fruit less the skin, core, and/or pit. Each fruit has a different formula for computing the prepared weight from the gross weight. In our example we will assume that the prepared weights of apples, oranges and plums are 85, 80, and 65 percent of gross weight (respectively). If we assume the weak interpretation of the Law then we get the following code.

```
(defmethod (FruitBasket :compute-weight) ()
   (send Fruits :compute-weight))

(defmethod (Fruits :compute-weight) ()
   (loop for each-fruit in child sum
      (send each-fruit :compute-weight)))

(defmethod (Apple :compute-weight) ()
   (* weight 0.85))

(defmethod (Orange :compute-weight) ()
   (* weight 0.80))

(defmethod (Plum :compute-weight) ()
   (* weight 0.65))
```

It is useful to note the use of the inherited instance variable weight within the :compute-weight methods attached to Apple, Orange, and Plum. This is in violation of the strong interpretation of the Law. The solution to this problem which is within the strong interpretation of the Law of Demeter would be as follows.

```
(defmethod (FruitBasket :compute-weight) ()
   (send Fruits :compute-weight))

(defmethod (Fruits :compute-weight) ()
   (loop for each-fruit in child sum
      (send each-fruit :compute-weight)))

(defmethod (Apple :compute-weight) ()
   (send self :get-percent-weight 0.85))

(defmethod (Orange :compute-weight) ()
   (send self :get-percent-weight 0.80))

(defmethod (Plum :compute-weight) ()
   (send self :get-percent-weight 0.65))

(defmethod (Fruit :get-percent-weight)
    (percent)
   (* weight percent))
```

The latter solution requires one extra method called :get-percent-weight which is attached to Fruit. This is to force the computation on the weight instance variable into a method which is attached to the Fruit class. This method is activated when either an Apple, Orange, or Plum object sends itself the :get-percent-weight message since those classes do not have this method name defined for them and they inherit from Fruit.

The two programs defined above both seem to work equally well. The advantage of the latter program will show up as we modify the underlying data structure. Let us assume that we will now expand our original problem. Many years have passed since we wrote the code and now mankind no longer finds itself earth bound. He has the ability to take fruit baskets to any number of different planets around the galaxy. Clearly the weight of an object can no longer be a simple number which represents the weight of the object on earth. Our notion of weight must include a calculation based on two factors, mass and gravity. Our underlying data structure will change to:

```
FruitBasket = Basket Fruits.
Fruits ~ { Fruit }.
Fruit : Apple | Orange | Plum
  *common* <weight> PlanetWeight.
PlanetWeight =
  <mass>  Number <gravity> Number.
Basket = . Apple = . Orange = . Plum = .
```

The only modified class in this class dictionary is Fruit. The Strong Law guarantees that only the methods attached to Fruit will need modifications. In this example we need only change the :get-percent-weight method.

```
(defmethod (Fruit :get-percent-weight)
    (percent)
  (send weight :get-percent-weight percent))

(defmethod (PlanetWeight :get-percent-weight)
    (percent)
  (* (* mass gravity) percent))
```

The set of methods written under the Weak Law require the modification of methods attached to Apple, Orange, and Plum. In this simple example these changes are not extensive, but consider problems with many alternatives (e.g. 100 different fruits) or a more complicated class dictionary.

The Trellis/Owl system [16] provides the concept of subtype-visibility which acts as a nice intermediate between the strong and weak interpretations of the Law. With the subtype-visibility concept they can fine-tune which operations (instance variables and methods) should follow the weak Law and which the strong Law. C++ [18] can achieve the same with private and protected data members.

9 Conforming to the Law

Given a method which does not satisfy the Law, how can we transform it so that it conforms to the Law? We outline an algorithm for transforming any object-oriented program into an equivalent program which satisfies the Law. In other words, we show that we can translate any object-oriented program into a "normal form" which satisfies the weak Law.

We assume first that the only variables used in a method are instance variables and arguments. We exclude local variables. The violation of the Law will happen in a nested method application.

```
(send ... (send
    (send (send b :m1) :m2)
        :m3) ...)
```

Here b is an instance variable or an argument. If b is not self we rewrite it as

```
(send b :n1) ...

(defmethod (B :n1) ()
  (send ...
    (send (send (send self :m1) :m2) :m3) ...)
```

Let the type of the object returned from method m1 be A1. We can rewrite the above method application as

```
(send (send self :m1) :m2-new)

(defmethod (A1 :m2-new) ()
  (send ... (send (send self :m2) :m3) ...)
```

The method m2-new has a nesting level which is by 1 smaller than the nesting level of the original nested method application. By repeatedly applying the transformations given, we can transform all the violations of the Law to the form

```
(send (send self :m1) :m2)
```

This is a helpful reduction in the complexity of studying the removal of bad style. Instead of having to consider all nested method applications, we have to study only double nesting. We distinguish between two cases.

- m1 is an instance variable access method (i.e. it is an explicit setting or retrieving of an instance variable value). The nested application is in good style by definition.
- m1 is not an instance variable access method. The violation is of the form:

```
(defmethod (C :M) ()
...
  (send (send self :m1) :m2)
...)
```

We rewrite it in good style in the following form:

```
(defmethod (C :M) ()
...
  (send self :M1 (send self :m1))
...)

(defmethod (C :M1) (arg :type Argtype)
  (send arg :m2))
```

When we allow local variables in our methods, we can use a similar transformation.

The transformations given above allow us to translate a given program in bad style into good style. There are two other, though less automatic, ways to achieve this goal which may help in arriving at more readable or intuitive code. These two techniques, called Pushing and Popping also may help in minimizing the number of arguments passed to methods and occurrences of code duplication. With lifting we lift a method up in the class hierarchy and with pushing we push it further down.

The above proof demonstrates that any object-oriented program written in bad style can be rewritten in a form which follows the Law of Demeter. However, such programs may contain messages to inherited instance variables, therefore violating the constraints of the Strong Law of Demeter which insists that only direct (non-inherited) instance variables be sent messages inside of a method. We present the following proof that any object-oriented program written to obey the Weak Law of Demeter can be automatically rewritten to obey the Strong Law of Demeter. This proof together with the previous proof guarantees that any object-oriented program written in poor style can be rewritten to obey the Strong Law of Demeter.

The difference between programs which follow the Weak Law and the Strong Law of Demeter is that those following the Weak Law may contain message sends to inherited instance variables. These may look like:

```
(defmethod (SubtypeClass :sample) ()
  (send price ':xyz))
```

where price is an inherited instance variable from a class called "InheritedClass" (from which SubtypeClass inherits). All such message sends can be automatically rewritten such that the offending message is sent to self and delegated to the inherited class as follows:

```
(defmethod (SubtypeClass :sample) ()
  (send self ':price-xyz))

(defmethod (InheritedClass :price-xyz) ()
  (send price ':xyz))
```

10 Compile Time Checking of the Law

The Law of Demeter (the original version, not the object version) can be checked at compile-time for useful subsets of object-oriented languages. Checking the Law is closely related to static type checking. When a language supports static type checking, we can enforce the Law. It is not necessary that the user gives a type to all the variables. This checking requires only that the class dictionary for the application be available at compile-time and that all the arguments and return values of methods are typed by the user.

In not strongly typed languages there are several cases which cannot be checked at compile-time.

- Class and/or Method Definitions at Run Time

  This occurs in some languages such as Lisp/Flavors and CLOS.

- Passing a Message as an Argument

  The following example demonstrates the passing of a method as an argument to another method. In the receiver method, the passed message is sent to an object "o".

  ```
  (defmethod (X :xyz) (m)
    (send (send o m) ':m2)))
  ```

  If we don't know the argument types and the result type of m we cannot check the Law at compile-time.

- Dynamic Message Name Calculation

  The following example demonstrates a case which cannot be tested at compile-time.

  ```
  (send
    (send o (concat ': some-string)) ':m)
  ```

  The string named "some-string" could be read in at run time in which case there is no way of knowing the signature of the message sent to "o".

11 Minimum Documentation

Since our primary goal is to produce guidelines for the production of software which can be easily maintained we should also consider how the software should be documented. The documentation of a method should include, as a minimum, the production which defines the class to which the method is attached. This production defines all the instance variable types. In addition, a method documentation should contain

- the types for each of the arguments
- the return types of methods and an indication whether the methods return newly created instances.
- the types of objects created by the method (directly or indirectly)

This documentation gives the reader of the method a list of types he or she has to know about for understanding the method and for following the Law.

12 Formulations of the Law

We give the formulation of the Law of Demeter ("object" version) in a few object-oriented languages Smalltalk-80 [4], CLOS [1], C++ [18], Eiffel [13]. Each formulation adapts the Law to the terminology of the particular language. For explanation and motivation see [10].

Smalltalk-80: For all methods M, and for all message expressions in M the receiver must be one of the following objects:

- **an argument object of M including objects in pseudo variables "self" and "super" or**
- **an instance variable object of the class to which M is attached.**

(Objects created by a method, or by methods which it calls, and objects in global variables are viewed as being passed by arguments.)

CLOS: For all methods M, all function calls inside M must use only the following objects as dynamic method selection arguments:

- **M's argument objects or**
- **a slot value of a dynamic method selection argument class of M.**

(Objects created by a method, or by functions which it calls, and objects in global variables are viewed as being passed by arguments. A dynamic method selection argument is an argument which is used for identifying the applicable methods at runtime.)
Note: This version of the Law is currently under debate by some CLOS developers and users.

C++: For all classes C, and for all member functions M attached to C, all objects to which M sends a message must be

- **M's argument objects, including the object pointed to by "this" or**
- **a data member object of class C.**

(Objects created by a member function, or by functions which it calls and objects in global variables are viewed as being passed by arguments.)

Eiffel: For all routines M, and for all calls of routines inside M the entity object must be one of the following objects:

- **an argument object of M or**
- **an attribute object of the class in which M is defined.**

(Objects created by a routine, or by routines which it calls, and objects in global variables are viewed as being passed by arguments.)

---

13 Conclusion

In this paper we have introduced a simple rule which we believe helps the production of structured and maintainable software. This rule, which we call the "Law of Demeter", encodes the ideas of data hiding and encapsulation in an easy to follow form for the object-oriented programmer. The Law can be easily checked at compile-time for a large class of object-oriented programs. The style of modular programming encouraged by the Law leads naturally to the construction of an interface between the application software and the implementation details of the class and object hierarchies. It is this interface which, together with the Law, enables the programmer to redesign the data structures and still leave most of the existing software intact. We require this level of flexibility in any scenario which exhibits a high rate of modification. Effectively reducing the impact of local changes to a software system can reduce many of the headaches of software maintenance.

We have seen that there is a price to pay. The greater the level of data hiding, the greater the penalties are in terms of the number of methods, speed of execution, number of arguments to methods and sometimes readability of the code. These trade-offs are clearly visible in the discussion of the weak and strong versions of the Law. The strong version implies greater data and information hiding at the cost of extra methods and method arguments. But in the long term these are not fatal penalities. Using "Modula-2 like" modules to collect related methods and definitions together helps significantly in organizing the increased number of smaller methods into maintainable packets. This facility along with the support of an interactive CASE environment can erase some of the penalties. The execution deficit can be removed by preprocessor or compiler technologies like inline code expansion or code optimization similar to the way tail recursion optimization is done at the moment.

In the past year, over one hundred undergraduate and graduate students scrutinized, challenged and tested the Law of Demeter. We have applied the Law in the ongoing development of the Demeter system (now over fourteen thousand lines of Lisp/Flavors code). Some very recent developments have also been some of the more intricate, e.g. a generic parser capable of parsing any input with respect to any class dictionary. The Law never prevented us from achieving our algorithmic goals (as guaranteed by the proof above) however it was sometimes the case that the methods had to be rewritten to comply with it. This task was not difficult and the results were generally more satisfying. In the context of the Demeter system which is an energetically growing system the Law is an invaluable asset. We will be using the Demeter system itself to automate the porting of the system to C++ and this task is made very much simpler because of the uniform, modular structure of the existing code.

We are continuing our investigation of modular object-oriented programming and we believe that the Law and its consequences will lead to the future development of "good" software. In addition, the Law allows us to consider a normal form for object-oriented programs which we hope will form a basis for the

development of object-oriented program verification techniques.

We are looking for companies who are interested in benefitting from Demeter technology. It is an easy "add-on" solution which allows you to continue to work with your favorite object-oriented system. The Demeter team can be reached electronically on CS net: lieber@corwin.CCS.northeastern.EDU

Acknowledgements We would like to thank Gar-Lin Lee for her feedback and contributions during the development of the ideas in this paper. Thanks also to Jing Na who, along with Gar-lin, tested the practicality of using the Law during the production of some of the Demeter system software. Mitch Wand pointed out the relationship between the Law and the Principle of Compositionality. He was also instrumental in initiating the investigation into the weak and strong interpretations. Carl Woolf suggested that the object version of the Law is the one to be followed conceptually. He provided the example we used to motivate the object version.

Members of the CLOS community (Daniel Bobrow, Richard Gabriel, Jim Kempf, Gregor Kiczales, Alan Snyder, etc.) have participated in the debate and/or formulation of the CLOS version of the Law.

References

[1] D. Bobrow, L. G. D. Michiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification. March 1988 1988. Draft submitted to X3J13.

[2] B. J. Cox. *Object-Oriented Programming, An evolutionary approach.* Addison Wesley, 1986.

[3] D. W. Embley and S. Woodfield. Assessing the quality of abstract data types written in Ada. In *International Conference on Software Engineering*, pages 144–153, IEEE, Singapore, 1988.

[4] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, 1983.

[5] J. Heijenoort. *From Frege to Godel.* Harvard University Press, 1967.

[6] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[7] T. Kaehler and D. Patterson. *A Taste of Smalltalk.* Norton, 1986.

[8] B. Kernighan and P. Plauger. *The Elements of Programming Style.* McGraw-Hill, 1974.

[9] K. Lieberherr. Object-oriented programming with class dictionaries. *Journal on Lisp and Symbolic Computation*, 1(2):pages unknown, 1988.

[10] K. J. Lieberherr and I. Holland. *Formulations of the Law of Demeter.* Technical Report Demeter-2, Northeastern University, June 1988. 12 pages.

[11] K. J. Lieberherr and A. J. Riel. Demeter: a CASE study of software growth through parameterized classes. *Journal on Object-Oriented Programming*, 1(3):pages unknown, 1988. extended version of paper with same title presented at the *10th International Conference on Software Engineering, Singapore*, pages 254-264.

[12] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development. The MIT Electrical Engineering and Computer Science Series*, MIT Press, McGraw-Hill Book Company, 1986.

[13] B. Meyer. *Object-Oriented Software Construction. Series in Computer Science*, Prentice Hall International, 1988.

[14] D. L. Parnas, P. C. Clements, and D. M. Weiss. Enhancing reusability with information hiding. In P. Freeman, editor, *Tutorial: Software Reusability*, pages 83–90, IEEE Press, 1986.

[15] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, 1985.

[16] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 9–16, 1986.

[17] A. Snyder. Inheritance and the development of encapsulated software systems. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 147–164, The MIT Press, 1987.

[18] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, 1986.