# Assuring Good Style for Object-Oriented Programs

Karl J. Lieberherr and Ian Holland
Northeastern University, College of Computer Science
Cullinane Hall, 360 Huntington Ave., Boston, MA 02115
lieber@ccs.neu.edu, phone: (617) 373 2077

**Abstract**

We introduce a simple, programming language independent rule (known in-house as the Law of Demeter) which encodes the ideas of encapsulation and modularity in an easy to follow form for the object-oriented programmer. The rule achieves the following related benefits if code duplication, the number of method arguments and the number of methods per class are minimized: Easier software maintenance, less coupling between your methods, better information hiding, methods which are easier to reuse, and easier correctness proofs using structural induction. We show relationships between the Law and software engineering techniques, such as coupling control, information hiding, information restriction, localization of information, narrow interfaces and 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.

## 1 Introduction

This paper describes the object-oriented programming style rule called *The Law of Demeter*. Along with the 'goto-rule' and other programming style rules inherited from the procedural programming paradigm, many of which still apply, the Law should be part of the programming knowledge that is considered when implementing object-oriented software. It is a partial response to the questions: "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 addition, it helps to formalize the existing ideas on these issues that can be found in the literature [KP86] [Sny87].

There are two kinds of style rules for object-oriented programming: rules that constrain the structure of classes and rules that constrain the implementation of methods. Style rules that influence the structure of classes have been published elsewhere [Lie88]. The focus of this paper is on a style rule that restricts how methods are written for a set of class definitions. In particular, the Law restricts the message-sending statements in method implementations.

Informally, the Law says that any object receiving a message in a given method must be one of a restricted set of objects. This set of *preferred* objects includes the method arguments, the

self pseudo-variable, and to the immediate subparts of self. The self object in Smalltalk and Flavors is called this in C++ and current in Eiffel.

The Law of Demeter is named after the Demeter System$^{TM}$, which provides a high-level interface to class-based object-oriented systems, and the Demeter Research Group at Northeastern University, which develops the system. The Group has applied the Law in the development of the system itself (formally about fourteen thousand lines of Lisp/Flavors and now about ninety thousand lines of C++ code) and in the implementation of numerous applications developed with the system.

Our experience has been that the Law promotes maintainability and comprehensibility of the software. This is a result of the small method size and the predicable message-passing patterns, both of which are caused by the application of the Law. In other words, following the Law in concert with rules such as, minimizing code duplication, minimizing the number of arguments, and minimizing the number of methods, produces code with a characteristic and manageable form.

We have also seen that adherence to the Law prevents programmers from encoding details of the class hierarchy structure in the methods. This is critical to the goal of making the code robust with respect to changes in the hierarchy structure. These changes occur very frequently in the early stages of development.

The goal of the Law of Demeter is to organize and reduce the *behavioral dependencies* between classes. Informally, one class behaviorally depends on another class when it calls a method (through a message sent to an object) defined in the other class. The behavioral dependencies encoded in the methods of an object-oriented program determine the complexity of the program's control flow and the level of coupling between the classes. This paper examines these relationships and illustrated how the Law impacts their existence.

Some other work describing the Law includes [LHR88] where we presented a proof which states that any object-oriented program written in bad style can be transformed systematically into a 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 *how* he or she solves it. We have also formulated interpretations of the Law for multiple programming languages [LH89b]. Third party commentary on the Law includes [Boo91, Sak88, Bud91, Gra91]. The thesis of Casais [Cas90] examines the Law in depth and assesses its favorable impact on the problem of providing automatic support for rewriting code in response to changes in the class hierarchy. A slight dissenting voice was raised by Wirfs-Brock et. al [WBW89] who prefer a function centered approach to object-oriented design rather than the data centered approach of Demeter.

The examples in this paper are written in the extended notation of the Demeter system. Section2 describes Demeter and its notation. The sections which follow will define the Law of Demeter both formally and through examples, examining both practical and theoretical issues.

# 2 Demeter

The key contribution of the Demeter system is to improve programmer productivity by several factors. This is achieved in a number of ways. First, Demeter provides a comprehensive standard library of utilities. Second, a significant amount of code is generated from the programmers object-oriented design. Third, Demeter includes a number of tools that automate common programming practices.

The key ideas behind the Demeter system are to use a more expressive class notation than in existing object-oriented languages and to take advantage of the added information by providing many custom-made utilities. These utilities are provided for a specific object-oriented language like C++ or Flavors and greatly simplify the programming task.

Examples of utilities Demeter generates or provides generically are: class definitions in a programming language, application skeletons, parsers, pretty printers, type checkers, object editors, re-compilation minimizers, pattern matchers and unifiers. The Demeter system helps the programmer define the classes (both their structure and functionality) with several support tools, including a consistency checker (semantic rules and type checking at the design level), a learning tool which learns class definitions from example object descriptions, an LL(1) corrector and an application-development plan generator [Lie88] [LR88]. The explanations and examples presented in this paper are written in the extended Demeter notation which is described below.

One of the primary goals of the Demeter system is to develop an environment that eases the evolution of a class hierarchy. Such an environment must provide tools for the easy updating of existing software (the methods or operations defined on the class hierarchy). We are striving to produce an environment that will let software be 'grown' in a continuous fashion. We believe a continuous-growth environment will lead to a rapid prototyping/system-updating development cycle.

The primary input to the system is a collection of class definitions. This collection is called a *class dictionary*. Classes are described in Demeter using three kinds of class definitions: *construction*, *alternation*, and *repetition*. The class dictionary shown in Figure 1 partially defines the structure of a lending library.[1]

1. A *construction* class definition is used to build a class from a number of other classes and is of the form

   **class** C **has parts**
   $partName_1 : SC_1$
   $partName_2 : SC_2$
   ...
   $partName_n : SC_n$
   **end class** C

   Here C is defined as being made up of $n$ parts (called its instance variable values), each part has a name (called an instance variable name) followed by a type (called an instance variable type). This means that for any instance (or member) of class C the name $partName_i$ refers to a member of class $SC_i$. The example shown in Figure 1 describes a library class as consisting of a reference section, a loan section, and a journal section.

   We use the following naming convention: instance variable names begin with a lower case letter and class names begin with an upper case letter.

2. An *alternation* class definition allows us to express a union type. A class definition of the form

[1] We use two notations in the Demeter system. This introductory paper uses the extended notation. A concise notation based on EBNF is used in later papers of the thesis. The abstract syntax of the concise and the abstract syntax of the extended notation are identical: only the "syntactic sugar" is changed.

```
class Library has parts
        reference : ReferenceSec
        loan : LoanSec
        journal : JournalSec
end class Library


class BookIdentifier is either
        ISBN or LibraryOfCongress
end class BookIdentifier


class ReferenceSec has parts
        refBookSec : BooksSec
        archive : Archive
end class ReferenceSec


class Archive has parts
        archMicrofiche : MicroficheFiles
        archDocs : Documents
end class Archive
```

```
class BooksSec has parts
        refBooks : ListofBooks
        refCatalog : Catalog
end class BooksSec


class ListofBooks is list
            repeat {Book}
end  ListofBooks

class Catalog is list
            repeat {CatalogEntry}
end  Catalog


class Book has parts
        title : String
        author : String
        id : BookIdentifier
end class Book
```

Figure 1: Library class dictionary

```
class C  is either
        A or B
end class C
```

states that a member of C is a member of class A or class B (exclusively). For example, the definition of BookIdentifier in Figure 1, expresses the notion that when somebody refers to the identifier of a book they are actually referring to its ISBN code or its Library of Congress code.

3. A *repetition* class definition is simply a variation of the construction class definition where all the instance variables have the same type and the programmer does not specify the number of instance variables involved. The class definition

```
class C is list
            repeat {A}
end  C
```

defines members of C to be lists of zero or more instances of A.

# 3 Forms of the Law

The Law of Demeter has two basic forms: the object form and the class form. The object form is the primary form. However, it is not possible to statically check code with respect to the

object form. The two versions of the class form are compile-time checkable approximations.

The two versions of the class form are called the *strict form* and the *minimization form*. The strict version rigorously restricts the dependencies between classes. However, in practice, it is difficult to completely adhere to the strict version. These potential 'law-breaking' situations are discussed below. The minimization version is the weakest expression of the Law and is phrased as a guideline rather than a strict rule. It allows additional dependencies between classes but asks the object-oriented programmer to minimize them and to document them by declaring special *acquaintance* classes.

## 3.1 Object form

The object version of the Law is based on the concept of *preferred supplier objects*. These are defined as follows:

> **Definition :** A **supplier** object to a method M is an object to which a message is sent in M. The **preferred supplier** objects to method M are:
>
> - the immediate parts of self or
> - the argument objects of M or
> - the objects which are either objects created directly in M or objects in global variables.

The programmer determines the granularity of the phrase "immediate subparts" of self for the application at hand. For example, the immediate parts of a list class are the elements of the list. The immediate parts of a "regular" class object are the objects stored in its instance variables.

In theory, every object is a potential supplier to any particular method. When a supplier object is sent a message in a method, the flow of control passes from the method to a method implemented for the message receiver. However, the presence of dynamic binding and method overriding in object-oriented programming languages can make it difficult to statically determine how control flows from one method to the next. By restricting the set of supplier objects we can contain the level of difficulty per method.

> **Definition :** *Object version of the Law of Demeter:* Every supplier object to a method must be a preferred supplier.

The object form expresses the spirit of the basic law and serves as a conceptual guideline for the programmer to approximate. While the object version of the Law expresses what is really wanted, it is hard to enforce at compile-time [LHR88]. The object version serves as an additional guide whenever the strict class version of the Law accepts a program which appears to be in bad style or when the strict class version of the Law rejects a program which appears to be in good style.

**Client**. Method M is a *client* of method f attached to class C if inside M message f is sent to an object of class C or to C. If f is specialized in one or more subclasses then M is only a client of f attached to the highest class in the hierarchy. Method M is a client of class C if it is a client of some method attached to C.

**Supplier**. If M is a client of class C then C is a *supplier* to M. Informally, a supplier class to a method is a class whose methods are called in the method.

**Acquaintance class**. A class C1 is an *acquaintance class* of method M attached to class C2, if C1 is a supplier to M and C1 is not

- the same as C2
- a class used in the declaration of an argument of M
- a class used in the declaration of an instance variable of C2

**Preferred-acquaintance class**. A *preferred-acquaintance* class of method M is either a class of objects created directly in M or a class used in the declaration of a global variable used in M.

**Preferred-supplier class**. Class B is called a preferred supplier to method M (attached to class C) if B is a supplier to M and one of the following conditions holds:

- B is used in the declaration of an instance variable of C or
- B is used in the declaration of an argument of M, including C and its superclasses, or
- B is a preferred acquaintance class of M.

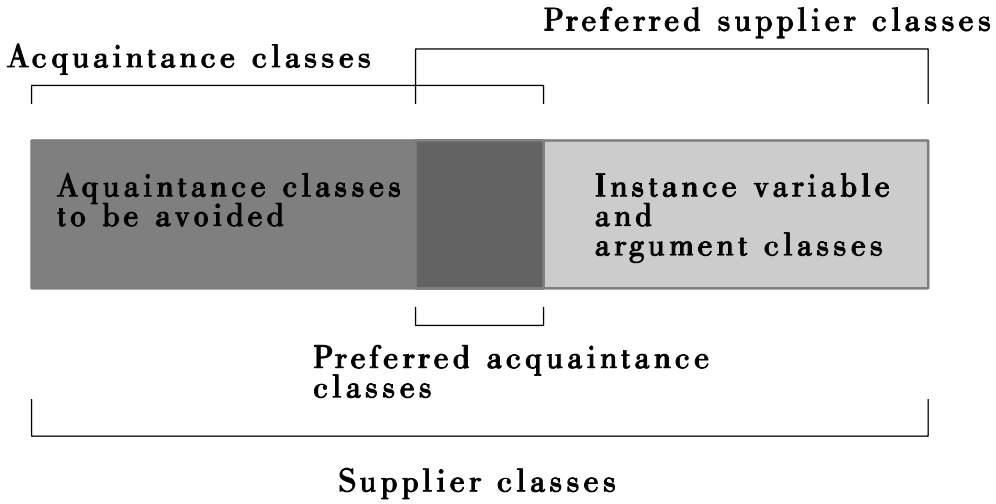


Table 1:

### 3.2 Class form

The class form's versions are expressed in terms of classes and can be supported by a compile-time law-enforcement tool. Paralleling the object form, the strict version is based on the notion of *preferred supplier* which is defined in table 1.

Figure 2 shows five examples of massages being sent to objects and of preferred-supplier classes. To send a message f to object s, we use the C++ notation (s → f() is the same as "send s the message f')'. In Figure 2 class B is a preferred supplier to method M and M is a preferred client of B. The ';' is the comment character which starts a comment line.

```
class C has parts
      s : B
      implements interface
          M() : Ident
          {calls s → f()}
          ; Ident is M's return type
end class C
```

Case 1: Instance variable class.

```
class C has parts
      ; none
      implements interface
          M(s : B) : Ident
          {calls s → f()}
end class C
```

Case 2: Argument class.

```
class B has parts
      ; none
      implements interface
          M() : Ident
          {calls self → f()}
          ; in C++ self is called this
end class B
```

Case 3: Argument class (self).

```
class C has parts
      ; none
      implements interface
          M() : Ident
              ; newObject is a new B instance
              {calls newObject → f()}
end class C
```

Case 4: Newly created object class.

```
class C has parts
      ; none
      implements interface
          M() : Ident
              {calls s → f()}
end class C
```

Case 5: Global class. s is global of type B.

In each case, class B is a preferred supplier to M

Figure 2: Examples of preferred suppliers.

As before, every class in an object-oriented program is a potential supplier of any method. However, it is best to limit the suppliers to a method to a small set of preferred classes. To define these preferred suppliers we introduce the concept of an *acquaintance class* of a method ([Sak88], [HB77]). A precise definition of an acquaintance relies on a class version of the supplier concept. Informally, a method's supplier class is a class whose methods are called in

the method.

The definitions make a distinction between the classes associated with the declaration of the method and the classes used in the body of the method. The former includes the class where the method is attached, its superclasses, the classes used in the declarations of the instance variables and the classes used to declare the arguments of the method. In some sense, these are an 'automatic' consequence of the method declaration. They can be easily derived from the code and shown by a browser. All other supplier classes to the method are introduced in the body of the method. They can only be determined by a careful reading of the implementation. This second set of classes are the acquaintance classes. To show these classes within a code browser would require a complete symbol table of the program.

The set of acquaintance classes are further partitioned into a *preferred acquaintance* subset and it's complement. A method's *preferred* acquaintance class is either a class of objects created directly in the method (by calling the acquaintance class's constructor) or a class used to declare a global variable used in the method.

Given these definitions, the strict version of the Law of Demeter's class form says:

> **Definition :** *Strict form of the Law of Demeter :* Every supplier class to a method must be a preferred supplier.

There are several benefits which result from applying the strict version of the Law's class form. For example, if the interface of class A changes, then only the preferred-client methods of class A and its subclasses require modification (provided that the changes required in the preferred client methods do not change the interfaces of those classes). A class's interface can change in many ways. For example, the programmer might modify an interface by changing an argument or return type, by changing the name of a method, or by adding or deleting a method. A class's preferred-client methods are usually a small subset of all the methods in a program; this reduces the set of methods that need to be modified. This benefit clearly shows that the Law of Demeter limits the repercussions of change.

Using the Law can also control the complexity of programming. For example, a programmer reading a method needs to be aware of only the functionality of the method's preferred supplier classes. These preferred suppliers are usually a small subset of the set all the classes in the application and furthermore, they are "closely related" to the class to which the method is attached. This relationship makes it easier to remember those classes and their functionality.

The second class version is more lenient than the strict form because it allows some non-preferred supplier classes. In practice, it makes sense to allow some of these other acquaintance classes. However, we suggest that the programmer clearly document the violations in order to recover the Law's benefits. Acquaintance classes are typically used for three reasons:

- Stability: If a class is stable and/or if its interface will be kept upwardly compatible, it makes sense to use it as an acquaintance class in all methods. The programmer specifies such "global" acquaintance classes separately and they are included in the acquaintance classes of all methods.
- Efficiency: The programmer might need to directly access instance variables of certain other classes to increase run-time efficiency. In C++ terminology, these are classes of which the method is a friend function.
- Object construction.

The permissive minimization version of the Law of Demeter is stated as follows:

> **Definition :** *Minimization form of Law of Demeter:* Minimize the number of acquaintance classes of each method.

We can count the number of acquaintance classes for all methods to assess the level of conformance of a program to the Law. If a class appears as an acquaintance class of several methods, it is counted as many times as it appears.

If a statically typed language like C++ or Eiffel is extended with a facility to declare acquaintance classes, the compiler can be modified in a straightforward way to check adherence to the minimization version in the following sense: Each supplier that is an acquaintance class must be explicitly declared in the list of the method's acquaintance classes. To easily check the Law at compile time or even at design time, the programmer must provide the following documentation for each method:

1. the types of each of the arguments and the result
2. the acquaintance classes.

The documentation gives programmers reading the method a list of the types they must know about to understand the method. The compiler can check the completeness of each method's documentation by examining the messages sent in the method and the classes of the objects created directly by the method.

# 4 Principles

The motivation behind the Law of Demeter is to ensure that the software is as modular as possible. The Law effectively reduces the occurrences of certain nested message sends (function calls) and simplifies the methods.

The Law of Demeter has many implications for widely known software engineering principles. Our contribution is to condense many of the proven principles of software design into a single statement that can be easily followed by object-oriented programmers and easily checked at compile-time.

Principles covered by the Law include:

- Coupling control. It is a well-known principle of software design to have minimal coupling between abstractions (like procedures, modules, methods) [EW88]. The coupling can be along several links. An important link for methods is the "uses" link (or call/return link) that is established when one method calls another method. The Law of Demeter effectively reduces the methods the programmer can call inside a given method and therefore limits the coupling of methods with respect to the "uses" relation. The Law therefore facilitates reusability of methods and raises the software's level of abstraction.
- 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, the programmer must use intermediate methods to traverse the "part-of" hierarchy in controlled small steps [LG86].

In some object-oriented systems, the programmer 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. The 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. [PCW85] [PCW86] on the modular structure of complex systems. To reduce the cost of software changes in their operational flight program for the A-7E aircraft they restricted the use of modules that provide information that is subject to change. 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 sends (function calls) by the Law of Demeter. Information restriction complements information hiding: Instead of hiding certain methods, they are made public but their use is restricted. Information restriction does not offer the same level of protection as information hiding. However, when hiding it is not feasible, restriction offers a level of protection.

- Localization of information.[2] Many software engineering textbooks stress the importance of localizing information. The Law of Demeter focuses on localizing *class* information. When programmers study a method they only have to be aware of types which are very closely related to the class to which the method is attached. They can effectively be ignorant (and independent) of the rest of the system. As the saying goes, 'ignorance is bliss'. This important aspect of the Law helps reduce programming complexity. In addition, the Law also controls the visibility of message names. Programmers can only use message names in the interfaces of the preferred-supplier classes to a given method.

- 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 [Hei67]. The main motivation behind the compositionality principle is that it facilitates structural induction proofs.

# 5 Example

This section shows how to apply the Law of Demeter to a program that violates both the strict and the minimization versions of the Law's class form. For this example, we use the classes defined by the class dictionary fragment for a library shown in Figure 3.

The methods of the example are written in C++. However, the text should be comprehensible for users of other object-oriented programming languages. In C++ terminology, a method is called a 'function member' and an instance variable is called a 'data member'. In the following C++ code, the types of data members and function member arguments are pointer types to classes.

The fragment of a C++ program in Figure 4 searches the reference section for a particular book. (To keep the example small, we use direct access to instance variables instead of using access methods.) The searchBadStyle function attached to ReferenceSec passes the message on to its book (BooksSec), microfiche (MicroficheFiles) and document sections (Documents).

This function breaks the Law of Demeter. The first message marked /**/ sends the message archMicrofiche to archive which returns an object of type MicroficheFiles. The method next sends

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

```
class Library has parts
        reference : ReferenceSec
        loan : LoanSec
        journal : JournalSec
end class Library


class BookIdentifier is either
        ISBN or LibraryOfCongress
end class BookIdentifier


class ReferenceSec has parts
        refBookSec : BooksSec
        archive : Archive
end class ReferenceSec


class Archive has parts
        archMicrofiche : MicroficheFiles
        archDocs : Documents
end class Archive
```

```
class MicroficheFiles has parts
        ...
end class MicroficheFiles


class Documents has parts
        ...
end class Documents


class BooksSec has parts
        ...
end class BooksSec
```

Figure 3: Library revisited

this returned object the search message. However, MicroficheFiles is not an instance variable or argument type of class ReferenceSec.

Because the structure of each classes is clearly defined by the class dictionary, the programmer might be tempted to accept the method searchBadStyle in Figure 4 as a reasonable solution. But 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:

```
class Archive has parts
        cdRomArch : CDRomFile
end class Archive


class CDRomFile has parts
        cdCSystem : ComputerSystem
        discs : CDRomDiscs
end class CDRomFile
```

The programmer now has to search *all* of the methods, including the searchBadStyle method, for references to an archive with microfiche files. It would be easier to limit the modifications only to those methods which are attached to class Archive. This is accomplished by rewriting the methods in good style resulting in searchGoodStyle functions attached to ReferenceSec and Archive.

Using good style also reduces the coupling respect to the "uses" relation: In the original

```
class ReferenceSec {
public:
   Archive* archive;
   BookSec* refBookSec;

   boolean searchBadStyle(Book* book) {
      return
         ( refBookSec → search(book) ||
   /**/  archive → archMicrofiche → search(book) ||
   /**/  archive → archDocs → search(book));
      }

   boolean searchGoodStyle(Book* book) {
      return
         ( refBookSec → search(book) ||
         archive → searchGoodStyle(book));
      }
   };


class Archive {
public:
   MicroficheFiles* archMicrofiche;
   Documents* archDocs;

   boolean searchGoodStyle(Book* book) {
      return
      (archMicrofiche → search(book) ||
      archDocs → search(book));
      }
   };
```

```
class MicroficheFiles {
public:
   boolean search(Book* book)
      {...}
};

class Documents {
public:
   boolean search(Book* book)
      {...}
};

class Book {....};
```

Figure 4: C++ fragment to search the reference section.

version, ReferenceSec was coupled with BooksSec, Archive, MicrofichеFiles and Documents, but now it is coupled only with BooksSec and Archive.

Another way to examine the effects of using the Law is to translate a program, in both good and bad style, into a dependency graph. In the graphs, the nodes of the graph are classes. An edge from class A to class B has an integer label which indicates how many calls are written in the text of the functions of A to the functions of B. If a label is omitted from an edge, it means that its value is 1. Access to an instance variable is interpreted as a call to read the instance variable. Figure 5a shows the graph for the program which violates the Law of Demeter; Figure 5b shows the graph for the one that follows the Law.

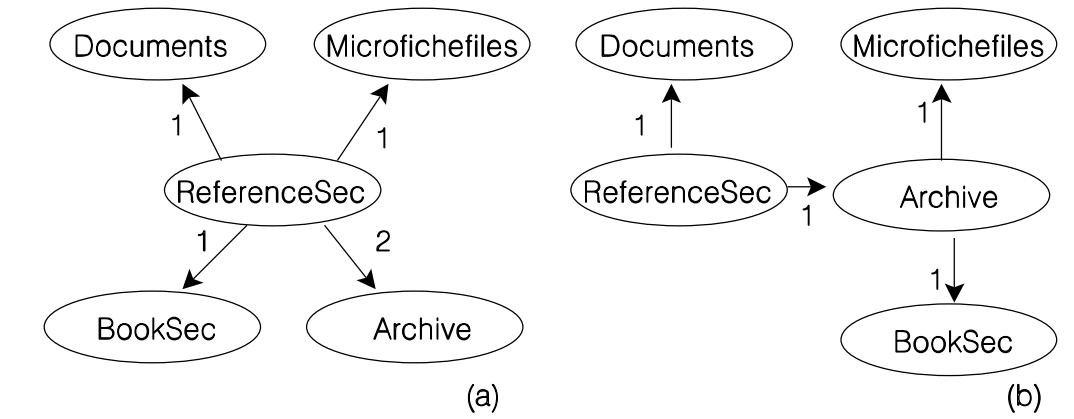


Figure 5: Dependency graph representation

# 6 Valid violations

The Law of Demeter is intended to act as a guideline, not as an absolute restriction. The minimization version of the Law's class form gives programmers a choice of how strongly they want to follow the Law: The more non-preferred acquaintance classes used, the weaker the adherence to the strict version. In some situations, the cost of obeying the strict version of the Law may be greater than the benefits. However, when programmers willingly violate the Law, they take on the responsibility of declaring the required acquaintance classes. This is critical documentation for future maintenance of the software.

As an example of where the cost of applying the Law is higher than its benefits, consider the following prototypical method which is in bad style, coded in both Flavors and C++:

*Flavors:*

```
(defmethod (C :M) (p)
   ( .... (send (send p :F1) :F2) ....))
```

*C++:*

```
void C::M(D* p)
   { ...; p → F1() → F2(); ... };
```

where p is an instance of class A and F1 returns a subpart of p. If the immediate composition of A changes the method M may have to change also because of F1.

There are two situations when it is reasonable to leave the above as it is:

- F1 is intended to serve as a "black box" and the programmer knows only about the types of its arguments and the return type. In this case, the maintainer of F1 has the responsibility to ensure that any updates to F1 are upwardly compatible so programmers of the function are not penalized for using it.
- If run-time efficiency is important to the application, the use of mechanisms such as the C++ friend function feature may be necessary. Friend functions should be used carefully, since whenever the private members of a class change, the friend functions of the class may also require change.

Consider another example that shows where the costs of using the Law might outweigh its benefits. For an application which solves differential equations the class dictionary may have the following definitions:

```
class ComplexNumber has parts
      realPart : Real
      imaginaryPart : Real
end class ComplexNumber
```

*Flavors:*

```
(defmethod (Vector :R) (c :ComplexNumber)
   ( .... (send (send c :realPart) :project self) ...))
```

*C++:*

```
void Vector::R(ComplexNumber* c)
   { ...; c → realPart → project(this); ... }
```

The method R is in the same form as M in the previous example and is in bad style for the same reason. The question here is whether it is important to hide the structure of complex numbers and to rewrite the method. In this application, where the concept of a complex number is well defined and well understood, it is unnecessary to rewrite the method so that the Law is obeyed.

In general, if the application concepts are well defined and the classes which implement those concepts are stable, in the sense that they are very unlikely to change, then such violations as the above are acceptable.

Our experience has been that writing programs which follow 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 increase in methods is related to the problem outlined in [LG86] 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 [LR88]. The functional abstraction is no longer a method but a module which will hide the lower-level methods.

# 7 Conforming to the Law

Given a method which does not satisfy the Law, how can a programmer transform it so that it conforms to the Law? In [LHR88] we described an algorithm to transform any object-oriented program into an equivalent program which satisfies the Law. In other words, we showed that we can translate any object-oriented program into a "normal form" which satisfies the Law's strict version.

There are other, less automatic, ways to achieve this goal which may help to derive more readable or intuitive code. These also may help to minimize the number of arguments passed to methods and the amount of code duplication. Two such techniques are called *lifting* and *pushing*.

To explain these techniques, we need a preliminary definition. We say that class B is a **part-class** of class A, if B is the class of one of A's instance variables or B is a part-class of a class of one of A's instance variables.

Consider the method:

*Flavors:*

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

*C++:*

```
void C::M()
   {this → m1() → m2();}
```

and T is the class of the object returned by m1. T is not a preferred supplier class of M. We distinguish two cases:

1. T is a part-class of C.
2. C is a part-class of T.

**Lifting.** This technique is applicable in the first case (T is a part-class of C). The idea is to make m1 return an object of an instance variable or argument class of C and adjust m2 accordingly. Method m2 is lifted up in the class hierarchy, from being attached to class T to being attached to an instance variable class of C.

For example, suppose a program is needed to parse an input using a grammar. A grammar is made up of a list of rules (productions) indexed by rule name. A fragment of the parse application is shown in Figure 6. This program fragment uses one acquaintance class (class Body in the method parse for Grammar).

The problem with the fragment is that method lookUp of Grammar returns an object of class Body which is not an instance variable class of Grammar. To transform the first method into good style, we must make the lookUp method return an instance of Rule and then adjust parseDetails. Figure 7 shows the modified version. The improved program fragment uses no acquaintance class.

But this lifting approach does not always work, consider Figure 8. This program fragment uses one acquaintance class (class Rule in method parse of Grammar). Here, we cannot transform the first method into good style by lifting the return type of the lookUp method.

**Pushing.** This technique is applicable in cases 1 and 2 (i.e. T is a part class of C and C is a part class of T respectively). The second case is slightly more complicated as it involves

```
class Grammar is list
        repeat {Rule}
end  Grammar


class Rule has parts
      body : Body
end class Rule
```

**C++:**

```
void Grammar::parse(Symbol* ruleName)
   {this → lookUp(ruleName) → parseDetails();}

Body* Grammar::lookUp(Symbol* ruleName)
   {...
   return rule → lookUp(ruleName) → getBody();
   }

void Body::parseDetails()
   { ...}
```

**Flavors:**

```
(defmethod (Grammar :parse) (ruleName :type Symbol)
   (send (send self ':lookUp ruleName) ':parseDetails))

(defmethod (Grammar :lookUp) (ruleName :type Symbol)
   ... (send (send rule ':lookUp ruleName) ':getBody))

(defmethod (Body :parseDetails) ()
   ...)
```

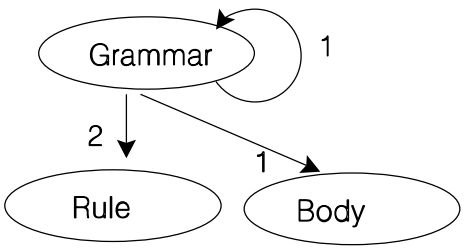


Figure 6: Example code that violates the Law of Demeter

*Flavors*

```
(defmethod (Grammar :parse) (ruleName :type Symbol)
   (send (send self ':lookUp ruleName) ':parseDetails))

(defmethod (Grammar :lookUp) (ruleName :type Symbol)
   ... (send rule ':lookUp ruleName))

(defmethod (Rule :parseDetails) ()
   ..(send self ':getBody) ...
)
```

*C++:*

```
void Grammar::lookUp(Symbol* ruleName)
   {this → lookUp(ruleName) → parseDetails();}

Rule* Grammar::lookUp(Symbol* ruleName)
   {...
   return rule → lookUp(ruleName);}

void Rule::parseDetails()
   {... this → getBody(); ...}
```

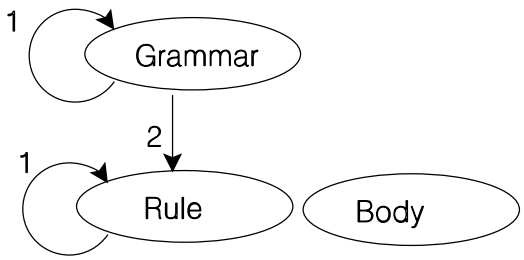


Figure 7: New parse implementation

```
class Grammar has parts
      ruleList : RuleList
end class Grammar


class RuleList is list
         repeat {Rule}
end  RuleList


class Rule has parts
      ruleName : Symbol
end class Rule
```

*Flavors*

```
(defmethod (Grammar :parse) (ruleName :type Symbol)
   (send (send-self ':lookUp ruleName) ':parseDetails))

(defmethod (Grammar :lookUp) (ruleName :type Symbol)
   ; returns object of type Rule
   ...
   (send ruleList ':lookUp ruleName))

(defmethod (RuleList :lookUp) (ruleName :type Symbol)
   .... )

(defmethod (Rule :parseDetails) ()
   .... )
```

*C++:*

```
void Grammar::parse(Symbol* ruleName)
   {this → lookUp(ruleName) → parseDetails();}

Rule* Grammar::lookUp(Symbol* ruleName)
   { ...
   ruleList → lookUp(ruleName);
   }

void RuleList::lookUp(Symbol* ruleName)
   { ... }

void Rule::parseDetails()
   { ... }
```

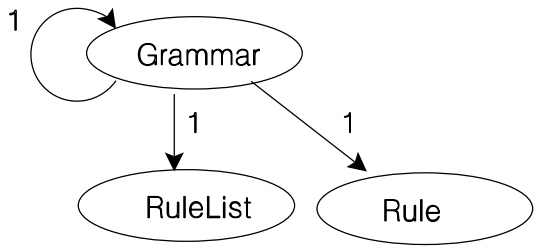


Figure 8: Law violation that cannot be fixed with the lifting technique.

traveling up the object hierarchy but the general technique is the same as for the first case. The pushing technique is just a variation of the top-down programming technique of pushing the responsibility for doing the work to a lower level procedure.

In the lifting example, a problem arose because the Grammar class has the task of sending the parseDetails message. This task is really the responsibility of class RuleList which knows more about Rule details than Grammar. Figure 9 shows an improved design that does not use any acquaintance classes. This is also the technique used in Figure 4 to write searchGoodStyle.

*Flavors :*

```
(defmethod (Grammar :parse) (ruleName)
   (send self ':lookUpParse ruleName))

(defmethod (Grammar :lookUpParse) (ruleName)
   (send ruleList ':lookUpParse ruleName))

(defmethod (RuleList :lookUpParse) (ruleName)
   (send (send-self ':lookUp ruleName) ':parseDetails))
```

*C++:*

```
void Grammar::parse(Symbol* ruleName)
   {this → lookUpParse(ruleName);}

void Grammar::lookUpParse(Symbol* ruleName)
   {ruleList → lookUpParse(ruleName);}

void RuleList::lookUpParse(Symbol* ruleName)
   {this → lookUp(ruleName) → parseDetails();}
```

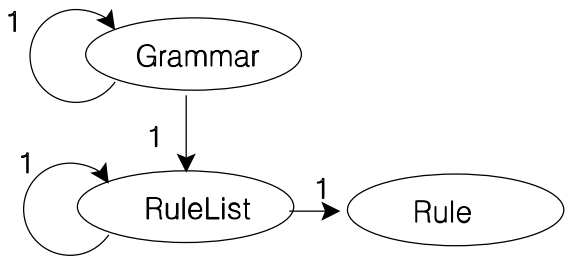


Figure 9: Example transformed with the pushing technique.

The redesign has introduced an additional method. If list classes are viewed as stable (for example, as is the case in Smalltalk), there is no need for the redesign and it is justified to keep the acquaintance class.

# 8 Conclusion

This paper introduced a simple rule which, when followed, results in the production of structured and maintainable object-oriented software. The rule, called the "Law of Demeter", encodes the ideas of data hiding and encapsulation in an easy to follow form for the object-

oriented programmer. The resulting code is more robust, allowing individual classes to be redesigned while leaving most of the remaining software intact. Furthermore, by effectively reducing the effects of local changes to a software system, adherence to the Law can reduce many of the headaches of software maintenance.

But following the Law exacts a price. The greater the level of interface restriction (a refinement of hiding), the greater the penalties are in terms of the number of methods, execution speed, number of arguments to methods and sometimes code readability.

But in the long term these are not fatal penalties. We have found that packaging the related methods and definitions together helps significantly in organizing the increased number of smaller methods [Lie92]. This facility along with the support of an interactive CASE environment can erase some of the penalties of following the Law. The Demeter System includes a formalism, and a code generation mechanism, called *Propagation Patterns* [LXSL91, LHSLX92] which removes most of the programming burden of following the Law. This utility generates major parts of the required code. The execution-speed problem can be countered by using preprocessor or compiler technologies like in-line code expansion or code optimization similar to the way tail recursion optimization is done.

In the application of the Law throughout the development of the Demeter System the Law never prevented us from achieving our algorithmic goals although some the methods needed to be rewritten. This task was not difficult and the results were generally more satisfying.

**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 was instrumental in initiating the investigation into the weak and strong interpretations. Carl Wolf suggested that the object version of the Law is the one to be followed conceptually. Special thanks are due to Arthur Riel who was a principal author on earlier versions of this paper.

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.

We would like to thank Markku Sakkinen for his interesting paper [Sak88] and his helpful mail messages about the Law of Demeter. Cindy Brown and Mitch Wand convinced us that we should use a more readable notation than EBNF and they helped us in designing it. Paul Steckler and Ignacio Silva-Lepe made several contributions to the extended Demeter notation. sectionBibliographic Note Earlier reports on the the work described in this paper have appeared as [LHR88, LH89b, LH89a].

# References

[Boo91] Grady Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings Publishing Company, Inc., 1991.

[Bud91] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison Wesley, 1991.

[Cas90] Eduardo Casais. Managing class evolution in object-oriented systems. In Dennis Tsichritzis, editor, *Object Management*, pages 133–195. Centre Universitaire D'Informatique, Genève, 1990.

[EW88] D. W. Embley and S.N. Woodfield. Assessing the quality of abstract data types written in Ada. In *International Conference on Software Engineering*, pages 144–153, Singapore, 1988. IEEE.

[Gra91] Ian Graham. *Object-oriented methods*. Addison-Wesley, 1991.

[HB77] Carl Hewitt and H. Baker. Laws for communicating parallel processes. In *IFIP Congress Proceedings*, pages 987–992. IFIP (International Federation for Information Processing), August 1977.

[Hei67] J.V. Heijenoort. *From Frege to Gödel*. Harvard University Press, 1967.

[KP86] Ted Kaehler and Dave Patterson. *A Taste of Smalltalk*. Norton, 1986.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Electrical Engineering and Computer Science Series. MIT Press, McGraw-Hill Book Company, 1986.

[LH89a] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[LH89b] Karl J. Lieberherr and Ian Holland. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices*, 24(3):67–78, March 1989.

[LHR88] Karl J. Lieberherr, Ian Holland, and Arthur 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*, pages 323–334, San Diego, CA., September 1988. A short version of this paper appears in IEEE Computer, June 88, Open Channel section, pages 78-79.

[LHSLX92] Karl J. Lieberherr, Walter Hürsch, Ignacio Silva-Lepe, and Cun Xiao. Experience with a graph-based propagation pattern programming tool. In *International Workshop on CASE*, Montréal, Canada, 1992. IEEE Computer Society.

[Lie88] Karl Lieberherr. Object-oriented programming with class dictionaries. *Journal on Lisp and Symbolic Computation*, 1(2):185–212, 1988.

[Lie92] Karl J. Lieberherr. Component Enhancement: An Adaptive Reusability Mechanism for Groups of Collaborating Classes. In J. van Leeuwen, editor, *Information Processing '92, 12th World Computer Congress*, Madrid, Spain, 1992. Elsevier.

[LR88] Karl J. Lieberherr and Arthur J. Riel. Demeter: A CASE study of software growth through parameterized classes. *Journal of Object-Oriented Programming*, 1(3):8–22, August, September 1988. A shorter version of this paper was presented at the *10th International Conference on Software Engineering, Singapore, April 1988, IEEE Press*, pages 254-264.

[LXSL91] Karl Lieberherr, Cun Xiao, and Ignacio Silva-Lepe. Propagation patterns: Graph-based specifications of cooperative behavior. Technical Report NU-CCS-91-14, Northeastern University, September 1991.

[PCW85] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, 1985.

[PCW86] David Lorge Parnas, Paul C. Clements, and David M. Weiss. Enhancing reusability with information hiding. In Peter Freeman, editor, *Tutorial: Software Reusability*, pages 83–90. IEEE Press, 1986.

[Sak88] Markku Sakkinen. Comments on "the Law of Demeter" and C++. *SIGPLAN Notices*, 23(12):38–44, December 1988.

[Sny87] Alan Snyder. Inheritance and the development of encapsulated software systems. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 147–164. The MIT Press, 1987.

[WBW89] Rebecca Wirfs-Brock and Brian Wilkerson. Object-oriented design: A responsibility-driven approach. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 71–76, New Orleans, LA, 1989. ACM.