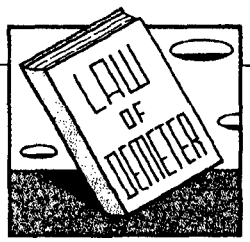


Assuring Good Style for Object-Oriented Programs

Karl J. Lieberherr and Ian M. Holland, Northeastern University

The language-independent Law of Demeter encodes the ideas of encapsulation and modularity in an easy-to-follow form for object-oriented programmers.

When is an object-oriented program written in good style? Is there some formula or rule that you can follow to write good object-oriented programs? What metrics can you apply to an object-oriented program to determine if it is good? What are the characteristics of good object-oriented programs?

In this article, we put forward a simple law, called the Law of Demeter, that we believe answers these questions and helps formalize the ideas in the literature.[1,2] There are two kinds of style rules for object-oriented design and programming: rules that apply to the structure of classes and rules that apply to how methods are written. Here, we focus on style rules that restrict how methods are written for a set of class definitions. We have published style rules for the structure of classes elsewhere.[3]

The Law of Demeter restricts the message-sending structure of methods. Informally, the law says that each method can send messages to only a limited set of objects: to argument objects, to the self pseudovariable, and to the immediate subparts of self. (The self construct in Smalltalk and Flavors is called "this" in C++ and "Current" in Eiffel.) In other words, each method depends on a limited set of objects.

The goal of the Law of Demeter is to organize and reduce dependencies between classes. Informally, one class depends on another class when it calls a function defined in the other class. We believe that the Law of Demeter promotes maintainability and comprehensibility, but to prove this in absolute terms would require a large experiment with a statistical evaluation. Because the field of object-oriented programming is relatively new, large object-oriented software developments that can provide data on the benefits of better handling dependencies are rare. However, we have examined our

0740-7459/89/0900/0038/$01.00 © 1989 IEEE

own code (about 14,000 lines of Flavors and C++ code) and are convinced of the law's benefits.

We developed the law during the design and implementation of the Demeter system, and so we named it after our system. Demeter provides a high-level interface to class-based, object-oriented systems. (Demeter is briefly described in the box on pp. 40-41; fuller descriptions appear in other papers.[3,4] The examples in this article are written in Demeter notation, which is explained in the same box.) Such a high-level interface supports an environment where code may evolve continuously rather than in sporadic jumps.

To achieve this continuous evolution, we would like programs to be well behaved or well formed in some sense. In other words, we would like the programs to follow a certain style that lets them be modified easily, minimizing changes required elsewhere in the programs. This ease of modification is one criterion that characterizes a good object-oriented programming style. Following the Law of Demeter will result in good style, provided the programmer follows other well-known style rules such as minimizing code duplication, minimizing the number of arguments, and minimizing the number of methods.

Every object-oriented programmer should know what is considered good object-oriented programming style, just as procedural programmers are aware of the top-down programming paradigm, the "thou shalt not use a goto" rule, and others. Many of the style rules for procedural programming are also applicable to object-oriented programming.

In an earlier paper,[5] we presented a proof that any object-oriented program 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 De-

Law of Demeter definitions

Client and supplier. A precise definition of the concept of acquaintance class relies on the concept of a client and supplier:

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. The exception is that if f is specialized in one or more subclasses then M is only a client of f attached to the highest class in the subclass hierarchy. Method M is a client of class C if M is a client of some method attached to class C.

Supplier. If method M is a client of class C as just described then C is a supplier of M. Informally, a supplier class of a method M is a class whose methods are called in M.

Acquaintance class. A precise definition of an acquaintance class is:

Class C_1 is an acquaintance class of method M attached to class C_2 if C_1 is a supplier to M and C_1 is not

- an argument class of M, including C_2, nor
- an instance variable class of C_2, nor
- a superclass of the above classes.

Informally, an acquaintance class of method M is a supplier class that is not an argument class of M nor an instance-variable class of the class to which M is attached.

Preferred-acquaintance class. A preferred acquaintance class of method M is either a class of objects created directly in M (by calling the acquaintance class's constructor) or a class of a global variable used in M.

Preferred-supplier class. Preferred supplier classes are defined formally in terms of preferred acquaintance classes:

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

- B is an instance-variable class of C or a superclass of such a class,
- B is an argument class of M, including C or a superclass of such a class, or
- B is a preferred acquaintance class of M.

Informally, the preferred supplier classes are made up of a method's preferred acquaintance classes and its instance-variable and argument classes. The relationships between supplier and acquaintance classes and their preferred subsets are given in Figure A.

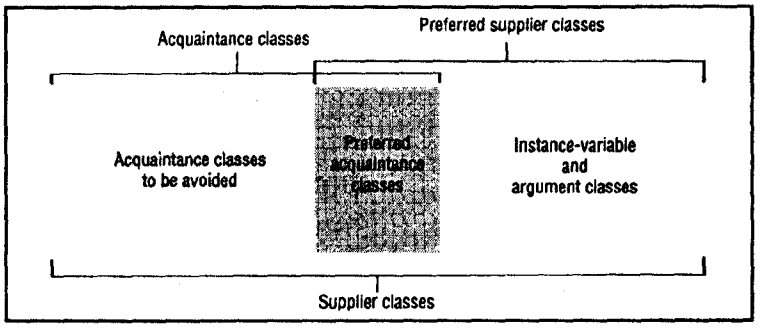


Figure A. Relationships between supplier and acquaintance classes and their preferred subsets.

meter does not restrict what a programmer can solve, it restricts only *how* he solves it.

We challenge object-oriented programmers to check if their programs follow our law and, where they do not, to consider whether they should. We believe all programmers who use object-oriented programming techniques should adopt our law. (The box on p. 47 describes instantiations of the Law of Demeter for common object-oriented languages.)

Forms of the law

The Law of Demeter has two forms: the class and object forms. The class form comes in two versions: the minimization and strict versions. The strict version is a special class form that rigorously restricts the dependencies between classes. The minimization version allows additional dependencies between classes but asks that you minimize them and document them by declaring acquaintance classes.

Class form. The class form's versions are expressed in terms of classes and can be supported by a law-enforcement tool.

Every class in an object-oriented design or program is a potential supplier of any method. However, it is best to limit a method's suppliers to a small set of preferred classes. To define these preferred supplier classes, we introduced the concept of an acquaintance class.[6,7] A precise definition of an acquaintance relies on the concept of a supplier; the box on p. 39 gives these precise definitions. Informally, a method's supplier class is a class whose methods are called in the method. A method's acquaintance class is a supplier class that is not an argument nor an instance-variable class. A method's *preferred* acquaintance class is either a class of objects created directly in a method (by calling the acquaintance class's constructor) or a class of global variables used in a method.

Acquaintance classes are typically used for three reasons:

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

Minimization version. The minimization version of the Law of Demeter's class form is the easiest to define:

> *Minimize the number of acquaintance classes over all methods.*

We count the number of acquaintance classes for all methods: If a class appears as

Demeter system overview

Demeter's key contribution is to improve programmer productivity by several factors for an important part of the development process: the preparation of a personalized software library for working with the objects defined by classes.

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 expressiveness by providing many custom-made utilities similar to a personalized library. 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 applies generically are class definitions in a language, application skeletons, parsers, pretty printers, type checkers, object editors, recompilation minimizers, pattern matchers, and unifiers. The Demeter system helps the user define the classes (both their structure and the high levels of their functionality) with several support tools, including a consistency checker (semantic rules and type checking at the design level), a learning tool that learns class definitions from example object descriptions, an LL(1) corrector for left-to-right scans with one look-ahead token producing a leftmost derivation, a script generator based on wish lists, and an application-development plan generator.

One of Demeter's primary goals 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 rather than in the sporadic jumps that undoubtedly lead to major rewrites. We believe a continuous-growth environment will lead to the fast-prototyping/system-updating development cycle common in the artificial-intelligence community.

Class definitions. Demeter describes classes with three kinds of class definitions: construction, alternation, and repetition. A collection of these class definitions is called a class dictionary. The class dictionary below partially defines a reference section of a library:

```
class ReferenceSec has parts
   ref_book_sec : BooksSec
   archive : Archive
end class ReferenceSec.

class Archive has parts
   arch_microfiche : MicroficheFiles
   arch_docs : Documents
end class Archive.

class BooksSec has parts
   ref_books : ListofBooks
   ref_catalog : Catalog
end class BooksSec.

class ListofBooks is list
   repeat {Book}
end class ListofBooks.

class Catalog is list
   repeat {Catalog_Entry}
end class Catalog.

class Book has parts
```

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, it is straightforward to modify the compiler to check adherence to the minimization version in the following sense: Each supplier that is an acquaintance class is declared in the list of the method's acquaintance classes.

To easily check the law at compile time or even at design time, the user must provide the following documentation for each method: (1) the types of each of the arguments and the result and (2) the acquaintance classes. The documentation gives the reader of the method a list of types he must know about to understand the method. The law-enforcement program must track the following additional information about each method: (1) the message sendings inside the method and (2) the classes of the objects created directly by the method.

Strict version. The strict version of the Law of Demeter's class form says:

> *All methods may have only preferred-supplier classes.*

These classes are made up of a method's preferred acquaintance classes and its instance-variable and argument classes. (Precise definitions are in the box on p. 39.) In essence, the strict version relies on restricting acquaintance classes.

Figure 1 shows five examples of preferred-supplier definitions. To send message *f* to object *s*, we use the C++ function-call notation ("*s*->*f*()" is the same as "send *s* the message *f*"). In Figure 1, class *B* is a preferred supplier of method *M*.

Applying the strict version of the law's class form has several benefits.

For example, if the interface of classes C_1 through C_n are changed, only the preferred-client methods of these classes require modification. A class's preferred-client methods are usually a small subset of all 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. We used a set of classes in this example benefit because changing the interface for a group of classes is a common task, one that is prompted by dependencies between interfaces.

You can change a class's interface in many ways. For example, you might modify an interface by changing an argument or return type, by adding or deleting an argument, by changing a name of a method, or by adding or deleting a method.

Using the law can also control the programming complexity. For example, when reading a method, you need be aware only of the functionality of the method's preferred supplier classes. These preferred suppliers are usually a small subset of all classes of the applica-

```
  title : String
  author : String
  id : BookIdentifier
end class Book.
```

A construction-class definition is used to build a class from several other classes and has the form

```
class C has parts
  part_name_1 : SC_1
  part_name_2 : SC_2
  ...
  part_name_n : SC_n
end class C.
```

An object of class *C* is defined as being made up of *n* parts (called its instance-variable values), and 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 element) of class *C* the name part_name_*i* refers to an element of class SC_*i*. The following example describes a library class consisting of a reference section, a loan section, and a journal section:

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

We use the following naming convention: Instance variable names begin with a lowercase letter and class names begin with an uppercase letter.

An alternation-class definition lets you express a union type. A class definition of the form

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

states that an element of *C* is an element of class *A* or class *B* (exclusively). For example,

```
class Book_Identifier is either
  ISBN or LibraryOfCongress
end class Book_Identifier.
```

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

A repetition-class definition is simply a variation of the construction-class definition where all the parts have the same type and you do not specify the number of parts involved. The class definition

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

defines elements of *C* to be lists of zero or more elements of *A*.

Notation. We use two notations in the Demeter system: A concise notation based on the EBNF extended Backus-Naur form and an expanded notation of our own that is largely self-explanatory. In this article, we use our expanded notation. The abstract syntax is identical for the concise and expanded notations: Only the syntactic "sugar" is changed.

```
;instance variable class:
class C has parts
  s : B
  implements interface
    M() returns Ident
      {calls s -> f()}
end class C.

;argument class:
class C has parts
  ; none
  implements interface
    M(s : B) returns Ident
      {calls s -> f()}
end class C.

;argument class:
class B has parts
  ; none
  implements interface
    M() returns Ident
      {calls self -> f()}
      ; in C++, self is called this
end class B.

;newly created object class:
class C has parts
  ; none
  implements interface
    M() returns Ident
      ; new_object is a new object of
      ; class B
      {calls new_object -> f()}
end class C.

; s is of type B, global
class C has parts
  ; none
  implements interface
    M() returns Ident
      {calls s -> f()}
end class C.
```

Figure 1. Examples of client and supplier definitions. Comment lines begin with a semicolon.

tion and, furthermore, are closely related to the class to which the method is attached. This relationship makes it easier to remember those classes and their functionality.

Object form. It is easy to extend a C++ compiler to check for the class form's strict version, but the price you pay for compile-time enforceability is that some programs that violate the spirit of the law will pass the test and that some programs that follow the spirit of the law will be rejected. The object version of the law says:

All methods may have only preferred-supplier objects.

This form expresses the spirit of the basic law and serves as a conceptual guideline for you to approximate in programming.

While the object version of the law expresses what is really wanted, it cannot be enforced at compile time. The object version serves as an additional guide in addition to the class version of the law.

The object form uses preferred-supplier objects, which are similar to preferred-supplier classes. A method's supplier object is an object to which a message is sent in that method. A method's preferred-supplier objects are

- the immediate parts of the pseudovariable self,
- the method's argument objects (which includes the pseudovariable self), or
- the objects that are either objects created directly in the method or objects in global variables.

The programmer determines the granularity of the "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 instance variables.

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 nested message sendings (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 easily be used by the object-oriented programmer and that can be 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, and methods). 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. The Law of Demeter effectively reduces the methods you can call inside a given method and therefore limits the coupling of methods for the Uses relation. The law therefore facilitates reusability of methods and raises the software's abstraction level.
- Information hiding. The Law of Demeter enforces one kind of information hiding[8]: structure hiding. The law generally prevents a method from directly retrieving a subpart of an object that lies deep in that object's Part-of hierarchy. Instead, you must use intermediate methods to traverse the Part-of hierarchy in controlled, small steps. 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 the law benefits even those systems with privacy: It promotes the idea that the instance variables and methods that are public should be used in a restricted way.
- Information restriction. Our work is related to the work by David Parnas and colleagues[9] 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 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 sendings by applying the Law of Demeter. Information restriction complements information hiding: Instead of hiding certain methods, you make them public but you restrict their use.
- Information localization. Many software-engineering textbooks stress the importance of localizing information, and the Law of Demeter focuses on localizing *type* information. When you study a method, you have only to be aware of classes that are very closely related to the class to which the method is attached. You 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. The law also

controls how visible message names are: In a method, you can use only message names that are in the interface of the preferred-supplier classes. This too localizes information.

• Structural induction. The Law of Demeter is related to the fundamental thesis of denotational semantics: 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,[10] which facilitates structural-induction proofs for program properties such as correctness with respect to a specification.

Example

To show how you can apply the Law of Demeter, consider 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 in Figure 2.

In C++, sending a message means calling a (virtual) member function. In the C++ examples, the types of data members and function arguments are pointer types to classes. Although the examples are in C++, the terms we use to explain the program can also be understood by users of Smalltalk and Flavors.

The fragment of a C++ program in Figure 3 searches the reference section for a book. (To keep the example small, we used direct access to instance variables instead of using access methods.) The search_bad_style function attached to ReferenceSec passes the message to its book (BooksSec), microfiche (MicroficheFiles), and documentation sections (Documents).

This function breaks the Law of Demeter. The first message marked /**/ sends the message arch_microfiche to archive, which returns an object of type MicroficheFiles. The method next sends this returned object the search message. However, MicroficheFiles is not an instance variable or argument type of class ReferenceSec.

Because the structure of all the classes are clearly defined by the class dictionary, you might be tempted to accept the method search_bad_style in Figure 3 as a reasonable solution, even though it violates the Law of Demeter. 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 videodiscs:

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

class ReferenceSec has parts
  ref_book_sec : BooksSec
  archive : Archive
end class ReferenceSec.

class Archive has parts
  arch_microfiche : MicroficheFiles
  arch_docs : 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 2. Class-dictionary fragment for a library.

```
class ReferenceSec {
public:
  Archive * archive;
  BooksSec * ref_book_sec;
boolean search_bad_style(Book* book) {
  return
      ( ref_book_sec->search(book) ||
/**/ archive->arch_microfiche->search(book) ||
/**/ archive->arch_docs->search(book));
      }
boolean search_good_style (Book * book) {
  return
      (ref_book_sec->search(book) ||
      archive->search_good_style(book));
  }
};
class Archive {
public:
  MicroficheFiles * arch_microfiche;
  Documents * arch_docs;
boolean search_good_style(Book* book) {
  return
      (arch_microfiche->search(book) ||
      arch_docs->search(book));
      }
};

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

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

class Book {
  ...
};
```

Figure 3. A C++ fragment to search the reference section for a book. Function search_bad_style violates the Law of Demeter.

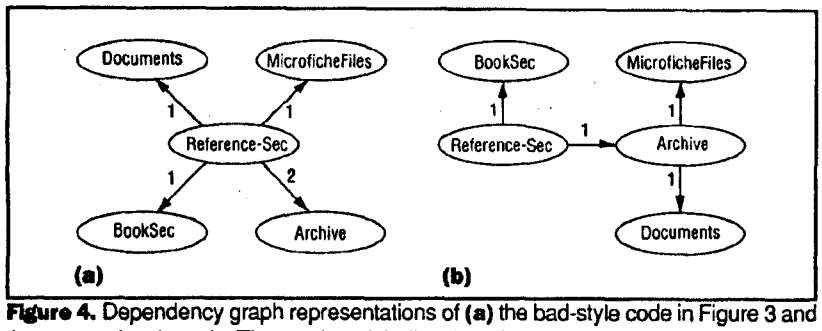


Figure 4. Dependency graph representations of **(a)** the bad-style code in Figure 3 and the **(b)** good-style code. The numbers labeling the edges between classes indicate the number of function calls from one class to functions of another class.

```
class Archive has parts
  cd_rom_arch : CD_ROM_File
end class Archive.

class CD_ROM_File has parts
  cd_c_system : ComputerSystem
  discs : CD_ROM_Discs
end class CD_ROM_File.
```

You now have to search *all* the methods, including the search_bad_style method, for references to an archive with microfiche files. It would be easier to limit the modifications only to those methods attached to class Archive. You accomplish this by rewriting the methods in good style, which results in search_good_style functions attached to ReferenceSec and Archive.

Using good style also reduces the coupling for the Uses relation: In the original version, ReferenceSec was coupled with BooksSec, Archive, MicroficheFiles, and Documents, but it now is coupled only with BooksSec and Archive.

Another way to examine the effects of using the Law of Demeter is to translate a program — in both good and bad style — into a dependency graph. In the graphs, the nodes are classes. An edge from class *A* to class *B* has an integer label that indicates how many calls that *A*'s functions make to class *B*'s. If a label is omitted from an edge, its value is 1. Access to an instance variable is interpreted as a call to read the instance variable. Figure 4a shows the graph for the program that violates the Law of Demeter; Figure 4b shows the graph for the one that follows the law.

Valid violations

The strict version of the class form of the Law of Demeter is intended to be a guideline, not an absolute restriction. The minimization version of the law's class form gives you a choice of how strongly you want to follow the strict version of the law: The more nonpreferred acquaintance classes you use, the less strongly you adhere to the strict version. In some situations, the cost of obeying the strict version may be greater than the benefits. However, when you willingly violate the law, you take on the responsibility of declaring the required acquaintance classes, which is useful documentation for future maintainers of your 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++. In Flavors, it is

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

In C++, it is

```
void C::M(A* 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 also have to change because of *F1*.

This is a situation when it is reasonable to leave this bad-style code 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* must ensure that any updates to *F1* are upwardly compatible so users of the function are not penalized for using it.

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

```
class Complex_Number has parts
  real_part : Real
  imaginary_part : Real
end class Complex_Number.
```

In Flavors, some code using these definitions would be

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

The same code in C++ is

```
void Vector::R(Complex_Number* c)
  {
    c -> real_part -> 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 that implement those concepts are stable, such violations are acceptable.

Writing programs that 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 that there can be too many operations in a type.[8] In this case, the abstraction may be less comprehensible, and implementation and maintenance more difficult, if you apply the law. There might also be an increase in the number of arguments passed to some methods.

Conformance

Given a method that does not satisfy the law, how can you transform it so that it conforms to the law? In an earlier paper,[5] we described an algorithm to transform any object-oriented program into an equivalent program that satisfies the law's strict version.

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

Using the following recursive definition, we use these techniques to transform

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

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

(a)

```
(defmethod (Grammar :parse) (rule_name)
  (send (send self :look_up rule_name) :parse_details))

(defmethod (Grammar :look_up) (rule_name)
  ...
  (send (send rule :look_up rule_name) :get_body))

(defmethod (Body :parse_details) ()
  ...
)
```

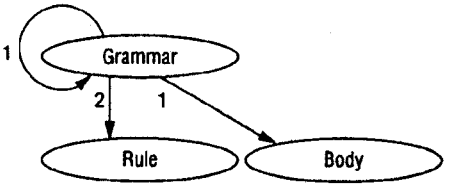
(b)

```
void Grammar::parse(Symbol* rule_name)
  {this -> look_up(rule_name) -> parse_details();}

Body* Grammar::look_up(Symbol* rule_name)
  {...
    return rule -> look_up(rule_name) -> get_body();
  }

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

(c)



(d)

Figure 5. Example code that violates the Law of Demeter: **(a)** class dictionary, **(b)** Flavors code, **(c)** C++ code, and **(d)** its dependency graph.

```
(defmethod (Grammar :parse) (rule_name)
  (send (send self :look_up rule_name) :parse_details))

(defmethod (Grammar :look_up) (rule_name)
  ...
  (send rule :look_up rule_name))

(defmethod (Rule :parse_details) ()
  ...
  (send self :get_body) ...)
```

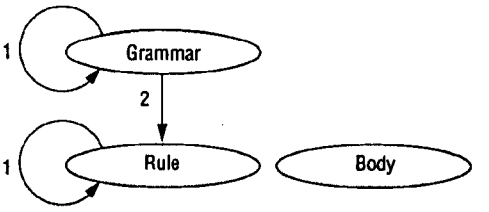
(a)

```
void Grammar::look_up(Symbol* rule_name)
  {this -> look_up(rule_name) -> parse_details();}

Rule* Grammar::look_up(Symbol* rule_name)
  {... return rule -> look_up(rule_name);}

void Rule::parse_details()
  {... this -> get_body(); ...}
```

(b)



(c)

Figure 6. Version of the Figure 5 example code transformed with the lifting technique to conform to the Law of Demeter: **(a)** Flavors code, **(b)** C++ code, and **(c)** its dependency graph.

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

class RuleList is list
  repeat {Rule}
end class RuleList.

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

(a)

```
(defmethod (Grammar :parse) (rule_name)
  (send (send-self :look_up rule_name) :parse_details))

(defmethod (Grammar :look_up) (rule_name)
; returns object of type Rule
  ...
  (send ruleList :look_up rule_name))

(defmethod (RuleList :look_up) (rule_name)
  ...
)

(defmethod (Rule :parse_details) ()
  ...
)
```

(b)

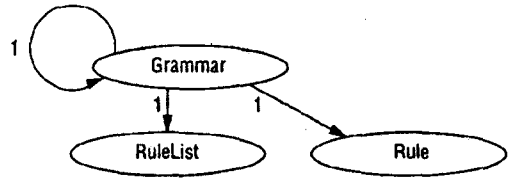
```
void Grammar::parse(Symbol* rule_name)
  {this -> look_up(rule_name) -> parse_details();}

Rule* Grammar::look_up(Symbol* rule_name)
  {...
    ruleList -> look_up(rule_name);
  }

void RuleList::look_up(Symbol* rule_name)
  {...}

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

(c)



(d)

Figure 7. Example code that violates the Law of Demeter and that cannot be fixed with the lifting technique: **(a)** class dictionary, **(b)** Flavors code, **(c)** C++ code, and **(d)** its dependency graph.

```
(defmethod (Grammar :parse) (rule_name)
  (send self :look_up_parse rule_name))

(defmethod (Grammar :look_up_parse) (rule_name)
  (send ruleList :look_up_parse rule_name))

(defmethod (RuleList :look_up_parse) (rule_name)
  (send (send-self :look_up rule_name) :parse_details))
```

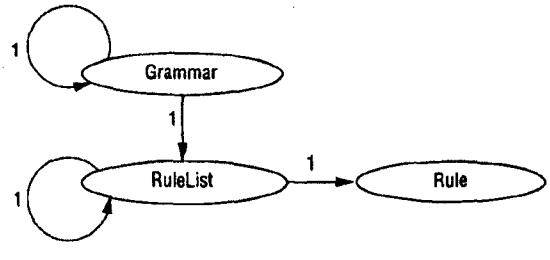
(a)

```
void Grammar::parse(Symbol* rule_name)
  {this -> look_up_parse(rule_name);}

void Grammar::look_up_parse(Symbol* rule_name)
  {ruleList -> look_up_parse(rule_name);}

void RuleList::look_up_parse(Symbol* rule_name)
  {this -> look_up(rule_name) -> parse_details();}
```

(b)



(c)

Figure 8. Version of the Figure 7 example code transformed with the pushing technique to conform to the Law of Demeter: **(a)** Flavors code, **(b)** C++ code, and **(c)** its dependency graph.

programs into forms that satisfy the Law of Demeter. The definition is that *B* is a part class of *A* if *B* is an instance-variable class of *A* or if *B* is a part class of an instance-variable class of *A*.

Consider the following program that violates the law's strict version. In Flavors, the method is

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

while in C++ it is

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

and *T* is the class of the object returned by *m*1. *T* is not a preferred supplier class of *M*. We distinguish two cases:

- *T* is a part class of *C*.
- *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 *m*1 return an object of an instance-variable or argument class of *C* and to adjust *m*2 accordingly. Method *m*2 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 you want to parse an input using some grammar. A grammar is made up of a list of rules such as that in Figure 5. This program fragment uses one acquaintance class (class Body in the method parse for Grammar) and is represented by Figure 5b and 5c.

The problem with the fragment is that method look_up of Grammar returns an object of type Body that is not an instance-variable type of Grammar. To transform the first method into good style, you must make the look_up method return an instance of Rule and then you must adjust parse_details. Figure 6 shows this modified version. This improved program fragment uses no acquaintance class.

But this lifting approach does not always work. Consider Figure 7. This program fragment uses one acquaintance class (class Rule in method parse of Grammar). Here, you cannot transform the first method into good style by lifting the return type of the look_up method.

Pushing. This technique is applicable in both cases (*T* is a part class of *C* and *C* is a part class of *T*, respectively). (The second case is slightly more complicated because it involves traveling up the object hierarchy, but the general technique is the same as for the first case.) It 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 parse_details message. This task is really the responsibility of RuleList, which knows more about Rule details than Grammar. Figure 8 shows an improved design that does not use any acquaintance classes.

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

Forms for popular languages

To use the Law of Demeter effectively, you must customize it to your language. Here are forms we have derived for several popular object-oriented languages. For C++, we give the class form's strict version; for the other languages, we give the object form of the law. This choice is arbitrary, but for the statically typed languages C++ and Eiffel, the class form is most useful because it can be checked by a modified compiler. Eiffel users will have little difficulty in formulating the class form.

C++, class form's strict version. In all member functions *M* of class *C*, you may use only members (function and data) of the following classes and their base classes:

- *C*,
- data-member classes of *C*,
- argument classes of *M*,
- classes whose constructor functions are called in *M*, or
- the classes of global variables used in *M*.

Common Lisp Object System, object form. We assume that the CLOS user can determine for each generic function the number of method-selection arguments (not necessarily all required ones) and that this number is part of the interface of the generic function. A method-selection argument is an argument used to identify the applicable methods.

All function calls inside method *M* must use only the following objects as method selection arguments:

- *M*'s argument objects,
- immediate parts of method-selection arguments of *M*, or
- an object that is either an object created directly by *M* or an object in a global variable.

Eiffel, object form. In all calls of routines inside routine *M*, the entity object must be one of the following objects:

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

Flavors, object form. In any method *M* attached to class *C* you may send messages only to the following objects:

- *M*'s argument objects,
- the instance variable objects of *C*, or
- an object that is either an object created directly by *M* or an object in a global variable.

Smalltalk-80, object form. In all message expressions inside method *M* the receiver must be one of the following objects:

- an argument object of *M*, including objects in pseudovariables Self and Super,
- an immediate part of Self, or
- an object that is either an object created directly by *M* or an object in a global variable.