Imposing The Law of Demeter and Its Variations

Partha pratim Pal* partha@cs.rutgers.edu

Naftaly H. Minsky† minsky@cs.rutgers.edu

Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

February 28, 1996

Abstract

The Law of Demeter [4] is accepted as a useful design principle that promotes tightly encapsulated classes and reduced coupling. Principles like this are routinely adopted in real-life projects, however neither the programming languages nor the existing environments provide enough support for effective realization of these principles. It is our thesis that broad structural principles should be formally specified, strictly enforced, and relaxed whenever relaxation is in order. In this paper we show how this can be done under our darwin-E environment using, the Law of Demeter as an illustration.

keywords: Law of Demeter, Law-Governed Architecture, Software Engineering Principles, Enforcement.

1 Introduction

One often adopts certain broad principles regarding the structure of a system under development. Such principles may reflect one's general programming style, a methodology adopted by the enterprise, or they may be part of the architecture of the system at hand. It is our thesis that in order for

*Supported by NSF grant No. CCR-9308773.

†Supported in part by NSF grant No. CCR-9308773, and in part by ARPA Contract Number DABT63-93-C-0064.

such principles to be reliably realizable, the following requirements must be satisfied:

1. The principles must be stated *explicitly* and *formally.*
2. The language for specifying the principles associated with a given system should be able to accommodate various *exceptions* to these principles.
3. The principles formulated for a given system, should be *enforced*, in order to ensure that they are actually satisfied by the system, throughout its evolutionary lifetime.

For a detailed discussion of this thesis and its implication under Law-Governed Architecture (LGA) [7], the reader is referred to [8], [9]. In this paper, we will illustrate our thesis by showing how a principle called the *Law of Demeter* (LoD, henceforth), proposed by Liberherr et al. in [4], is established under the darwin-E environment, that supports LGA.

We start this paper with a brief discussion of the Law of Demeter, which attempts to reduce the dependency between classes by restricting the ability of any given class to access certain features of other classes. In Section 3 we present an overview of the concept of law-governed architecture (LGA) and of the darwin-E software development environment that supports it. In Section 4 we show how LoD can be imposed on a system developed under darwin-E. In Section 5 we argue that the uniform imposition of LoD is sometimes unnecessary, and even detrimental, and show how to accommodate various exceptions to LoD under darwin-E. Section 6 briefly mentions other attempts to impose the Law of Demeter. Section 7 concludes the paper.

2 The Law of Demeter

The *law of Demeter* is a design principle that *rejects* certain operations from inside the methods of a class. One of the formulations of the law which lends itself to compile-time enforcement can be phrased as follows[1]:

The Law of Demeter (Class Form): *Inside an operation* `o` *of class* `c` *one should call operations of only the following classes (that are often called the preferred supplier classes):*

1. *the classes of the immediate subparts (computed or stored) of the current object,*
2. *the classes of the argument objects of* `o` *or the class itself,*
3. *the classes of objects created by* `o`*.*

In essence, if operation `o` of class `c` contains a call to another class `d`, such that `d` is not mentioned in the feature declaration part of `c`, or in the declaration of formal arguments or local entities of the operation `o`, then such a call violates the law of Demeter. As an illustration, consider the following code fragment which is a translation of the C++ code fragment in page 15 of [5] into Eiffel:

```
class BOOK
feature
      ...
      ...
end; -- class Book

class MICROFICHEFILES
feature
```

[1]Different formulations, with minor differences can be found in [6, 4, 5]. This particular version, obtained from Karl Lieberherr by private communication, is the most recent formulation.

```
   search(book: BOOK):BOOLEAN is
          ...
          ...
   end;
end; -- class MicroFicheFiles

class DOCUMENTS
feature
   search(book: BOOK):BOOLEAN is
          ...
          ...
   end;
end; -- class Documents

class ARCHIVE
feature
   m: MICROFICHEFILES;
   d: DOCUMENTS;
   search_good_style(book:BOOK): BOOLEAN is
   do
      Result:= m.search(book) or d.search(book)
   end;
end; -- class Archive

class REFERENCESEC
feature
   a: ARCHIVE;
   search_bad_style(book: BOOK): BOOLEAN is
   do
      Result:= a.m.search(book) or a.d.search(book)
   end;

   search_good_style(book: BOOK): BOOLEAN is
   do
      Result:= a.search_good_style(book);
   end;
end -- class ReferenceSec
```

The function `search_bad_style` in class `REFERENCESEC` violates the law of Demeter because it calls the `search` function on classes `MICROFICHEFILES` and `DOCUMENTS`, that do not appear anywhere in the body of the class

REFERENCESEC. In other words, REFERENCESEC is not directly involved with DOCUMENTS or MICROFICHEFILES, however the function search_bad_style induces a dependency on these two classes. Such dependencies are considered harmful and are not approved by LoD.

LoD controls the complexity of programming by restricting the dependency of a given class c to a set of classes that are "closely related" to it: the classes it refers to as *preferred suppliers*. The leverage comes from the fact that this set is a small subset of all classes in the application. To illustrate this let us consider a method m1 of a class c1 being renamed. Without LoD, m1 can potentially be called from any class in the system, and therefore such a change can break the system in hard to locate places. If LoD is followed, a class c can call m1 only if c1 is "closely related" to c, a fact that can be determined by examining the class c itself. Therefore, adapting to such changes becomes easier. Similarly, if the composition of c1 changes (however the public interface stays invariant), LoD ensures that only c1 and its children are affected. In [6],[4],[5] these and other benefits of LoD are discussed in great detail. LoD helps in systematizing reuse as well: when class c is being reused in a different context only this small subset of "closely related" classes are needed to be brought in. Then for each class in this subset one can recursively compute which other classes need to be brought in as well. However, the notion of "closely related" classes postulated as the preferred suppliers in LoD is not arbitrary, it has been shown in [5] that the calls permitted by LoD are *necessary* in the sense that if we disallow any of these calls in an attempt to reduce the dependency on other classes any further, we will not be able to write useful programs.

Despite the demonstrated benefits of LoD, there is very limited support for it from the existing technology. It is not enforced by any programming language, and very few tools exist (refer to Section 6) for checking classes against LoD. Use of this principle, is therefore limited to posing it as a *guideline* on class design and implementation. We argue that such principles should be *imposed* in order to be effective and as we shall see, LGA and darwin-E provide the necessary mechanism for doing so.

3 Overview of Darwin-E: an Environment Supporting LGA for Eiffel

In this section we present a partial overview of the darwin-E environment and introduce LGA concepts in terms of darwin-E to the extent necessary for understanding this paper. For further information on LGA refer to [7], and details of Darwin-E will be presented in a forthcoming paper [10].

Darwin-E is an SDE (Software Development Environment) that supports LGA for software written in Eiffel. The main novelty of LGA is that it associates with each project $\mathcal{P}$ a law $\mathcal{L}$ that governs various aspects of $\mathcal{P}$. Accordingly, the law of a project under darwin-E has the following, among others, under its jurisdiction:

- the actions of the developers participating in the project,
- the interactions between the Eiffel classes that are part of the system developed in the project, and
- the evolution of the law itself.

In this paper we will mostly be concerned about controlling inter-class interactions by law, and consequently, we will often use *law* or *law of the system* to refer to the subset $\mathcal{L}$ that control these.

According to LGA formalism, various entities involved in software project, such as, the software components (Eiffel classes in case of Darwin-E), software configurations and even software developers are represented as various objects in Darwin-E. Objects are associated with various properties (often referred to as attributes) that are stored as Prolog [1] terms in the object. In this paper we will be concerned with only one type of objects, those representing Eiffel classes and their properties.

Each Eiffel class is represented in Darwin-E by what is known as a module-object. For brevity, we will assume in this paper that the name of a module-object and the class it represents are identical. This means, depending on the context, the identifier `c` may mean the module-object named `c` as well as the Eiffel class attached to the module-object `c`.

Properties of darwin-E objects can be broadly categorized into *built-in properties* and *user-defined properties*. The semantics of the built-in properties are defined by darwin-E, and the semantics of the user-defined properties are defined by the law of the project. Likewise, module objects do have both kinds of properties, among which only some are relevant for this exposition.

The built-in property `eiffelSource(f)` in a module-object refers to the fact that it is associated with the code stored in the file `f`. Each module-object has a set of built-in properties that are obtained by darwin-E by analyzing the code associated with it. These properties collectively represent the interface of the class. For example, the module-object `c1` will contain the property `defines(attribute(x),type(c2))`, if `c1` defines an entity `x` of class `c2` as a class attribute. Similarly, if a class `c1` inherits from another class `c3`, a `inherits(c3)` property will be present in module-object `c1`.

User-defined properties of module objects are used to classify them and to denote information relevant to the law. For example, the term *cluster* is used in Eiffel literature to denote a set of closely related classes. The property `cluster(l)` can be associated with each module that represents a class in the cluster `l` to denote its membership to the cluster `l`. This property then, can be used in the law to distinguish between classes that are in cluster `l` and others. We will introduce other such properties whenever we need them in course of our discussion.

In darwin-E, the code of an Eiffel system $\mathcal{S}$ is analyzed statically by a compilation pre-pass to identify the interactions among the classes that constitute $\mathcal{S}$. Such interactions are then evaluated against the law to see if there is any interaction that violates the law. If a violation is detected in this law-enforcement step, assembly and running of the system is prevented and documentation about the violation is provided.

In order to impose the law of Demeter, we need to consider the *call interaction* only, but there are many others (refer to [9] for more on controllable Eiffel interactions) such as *inherit*, *being a client* etc, which are beyond the scope of this exposition.

A *call interaction* is defined as follows:

Definition 1 (call interaction) *Consider a routine* `r` *of class* `c1` *and a feature* [2] `f` *originating in class* `c2` *(see Definition 2 for what we mean by a feature being originated in a given class.). We say that the interaction* `call(r,c1,f,c2)` *occurs, if the feature* `f` *of* `c2` *is called ¿from routine* `r` *of class* `c1`*.*

Definition 2 (feature originating in a class) *A feature* `f` *of class* `c1` *is said to be originated from class* `c2` *if* `c2` *is the closest ancestor of* `c1` *where* `f` *has been defined, redefined or renamed.*

The law of a project under darwin-E consists of several Prolog clauses known as the *rules*[3]. The rules together form a Prolog program that is invoked during law-enforcement. For instance, this is what happens to a

[2] In Eiffel, the term feature is used to denote both methods and instance variables. When needed we will use routines (for methods) and attributes (for instance variables) to distinguish between features.

[3] The syntax of a rule is similar to that of Prolog clauses (`head:- body.`) with a few extensions. The term `p@t` in the body of a rule that tests the presence of the property `p`

typical call interaction `call(r,c1,f,c2)` during law-enforcement. First, the goal `cannot_call(r,c1,f,c2)` is posed to the law. If this goal succeeds, we say that the call is prohibited by law and flag a violation. Otherwise, the goal `can_call(r,c1,f,c2)` is posed to the law. If this succeeds, the call is accepted as a legal one. If the `can_call(r,c1,f,c2)` fails on the other hand, we say that the call is not permitted by the law and flag a violation. Enforcement proceeds to the next interaction once the fate of the current one: either legal or a violation, is decided. Therefore, call interactions can be controlled by defining permission rules of the form

```
can_call(...):- ...
```

and/or prohibition rules of the form

```
cannot_call(..):- ...
```

in the law, where prohibition rules always takes a higher precedence than the permission rules. Some calls are deliberately kept outside the scope of such control. Of interest to us in this paper, are the following two types of call interactions, which are always considered legal: 1) a class calling itself and 2) calls made via universal features obtained by the implicit inheritance from the class `any`.

In this paper we will only use permission rules for controlling call interactions, which means that the law of a project will contain

```
can_call(..):- ..
```

rules only (Refer to Figure 1 for examples) and if a call interaction is not accepted by these rules, it will not be allowed.

in the object `t`, for example, is one such extension that will see in the example rules. In the rules, upper-case and underscore (_) arguments denote variables that could be unified with constants such as individual objects. Individuals are shown in lower-case letters.

4 Uniform Imposition of the Law of Demeter under Darwin-E

In order to impose LoD uniformly over all the classes developed in a project under darwin-E, we need to incorporate it in the law $\mathcal{L}$ of the project. In this section we will formulate LoD in terms of a set of darwin-E rules which can be included in $\mathcal{L}$. The three clauses in the statement of LoD as presented in Section 3, results in four `can_call` rules, that are presented in Figure 1, annotated with *italicized* comments. The correspondence between the clauses in the statement of LoD and our rules is explained below.

$\mathcal{R}$1.
```
can_call(_,C1,_,C2) :-
        defines(attribute(_),type(C2))@C1 |
        defines(routine(_),type(C2))@C1.
```
Calls made to classes of features declared in the class are allowed.

$\mathcal{R}$2.
```
can_call(_,C1,_,C2) :-
        ( defines(attribute(_),type(C2))@C3 |
        defines(routine(_),type(C2))@C3 ),
        heirOf(C1,C3).
```
Calls made to classes of features declared in `C1`*'s ancestors are allowed. The goal* `herirOf(_,_)` *is defined by a built-in darwin-E rule with obvious meaning.*

$\mathcal{R}$3.
```
can_call(R,C1,_,C2) :-
        defines(_,of(C2),as_arguments_of(R))@C1.
```
Calls to the classes of arguments of routine `R` *are allowed from* `R`.

$\mathcal{R}$4.
```
can_call(R,C1,_,C2) :-
        defines(_,of(C2),local_to(R))@C1.
```
Calls to classes of local entities are allowed.

Figure 1: Rules That Impose LoD

Clause 1 of LoD deals with the *stored and computed* immediate subparts of an object. To see what it means in terms of Eiffel, consider an instance `i` of a class `c`: the attributes of `c` are the *stored* immediate subparts of `i` and the routines of `c` that return objects, are the *computed* subparts of `i`. In rule $\mathcal{R}1$, calls to classes of the features (attributes and routines) declared in `c` are permitted. The term `defines(attribute(_),type(C2))@C1` in rule $\mathcal{R}1$ tests if the class `C1` has an attribute of class `C2` and similarly, the term `defines(routine(_),type(C2))@C1` tests if the class `C1` defines a routine that returns an object of class `C2`. This rule alone, however, may not enough for the first clause, because of the choice whether or not the inherited features are considered to be *immediate subparts*. If we choose to consider inherited features to be immediate subparts, an interpretation which is often known as the weak version of the law of Demeter, we will need Rule $\mathcal{R}2$. This rule approves calls made to classes of features declared in ancestor classes. If we choose the other alternative, known as the strong version of LoD, which does not consider inherited features as immediate subparts of an objects, we will not need Rule $\mathcal{R}2$.

Clause 2 of LoD allows calls from a routine `r` of class `c` to `c` itself and the classes of the arguments of `r`. Rule $\mathcal{R}3$ allows calls from a routine to the classes of its arguments. The term `defines(_,of(C2),as_arguments_of(R))@C1` in this rule tests if `R` has a formal argument of class `C2`. In darwin-E calls to a class itself are always legal, therefore we do not need to worry about such calls in our rules.

Clause 3 of LoD allows calls to the classes of objects created by a routine. In Eiffel remote creation is not allowed, a routine `r` of class `c` can create only local entities or attributes of class `c`. Classes of attributes are already dealt with in rule $\mathcal{R}1$, and rule $\mathcal{R}4$ permits calls to the classes of the local entities of `r`.The term `defines(_,of(C2),local_to(R))@C1` in rule $\mathcal{R}1$ tests if `R` defines a local entity of class `C2`.

5 Controlled Relaxation of the Law of Demeter

Unfortunately, uniform imposition of LoD is not always desireable. Like other broad principles, it is not always useful, and is sometimes detremental. In this section we describe several such situations, and show how LoD can be relaxed to accommodate them by introducing exceptions in the law of the system. We conclude this section with a brief note on how the process of admitting exceptions is controlled under darwin-E.

5.1 The Case of Transient Classes

If a class is used only for a small transitory period of time, its dependencies on others are not important, because we will neither reuse it nor maintain it as the other classes evolve. One might want to relax LoD so that it does not apply to such classes. This can be done as follows.

Given a class `c`, which we consider transient, one can add the following rule to law $\mathcal{L}$:

$\mathcal{R}5$. `can_call(_,c,_,_) :- true.`

This rule simply permits an individual class `c` to make any call.

One can also systematize this concept of *transient* classes by characterizing all classes considered transient by a property `status(transient)`. In this case one can relax LoD for all these classes by the following single rule:

$\mathcal{R}6$. `can_call(_,C,_,_) :- status(transient)@C.`

5.2 The Case of Stable Classes

A certain set of classes in a project may be very stable and unlikely to change. Examples of such classes are the classes in installation library or classes that contain legacy code. Constraining calls to these classes by LoD does not serve any purpose since the callee classes are not likely to change and dependency on such classes are therefore not harmful as far as LoD is concerned. One can characterize all such classes by means of the property `status(stable)`, and add the following rule to the law of the project, in order to admit all calls made to a *stable* class:

```
R7. can_call(_,_,_,C) :- status(stable)@C.
```

Classes in the installation library are a special case of these *stable* classes. In darwin-E, any class from the Eiffel installation library that is used in the system, is already characterized by the built-in property `library`. This property can be utilized in admitting all kinds of calls to library classes from any class in the system in a manner similar to rule $\mathcal{R}7$:

```
R8. can_call(_,_,_,C2) :- library@C2.
```

5.3 The Case of a Close-Knit Cluster

Some clusters of classes may have a very cohesive structure with strong interdependencies. In this case preventing classes in the same cluster to call each other just because they violate the law of Demeter may not make sense, since any change in one class is likely to change all the classes in the cluster anyway. Moreover, the prohibition of LoD on inter-cluster calls can be particularly annoying in this case, because they are likely to be used very often and by the prorgrammers (the ones who designed and maintains this cluster) who know what they are doing. In this situation, one might include the following rule in the law:

```
R9. can_call(_,C1,_,C2) :-
        cluster(X)@C2,
        cluster(X)@C1.
```

This rule exempts the calls made by a class to another class in the same cluster from LoD.

5.4 The Case of Acquaintance Classes

The notion of *acquaintance classes* of a method was proposed in [5]. A class `c1` may have an unusually close relationship with a class `c2`, although in a routine `r` of `c1`, calls to `c2` do not satisfy LoD. In this case it makes sense to allow such calls. To incorporate this we may use the following rule:

$\mathcal{R}10$. `can_call(r,c1,_,c2) :- true.`

This notion can be extended to make `c2` an acquaintance of all routines of `c1` as follows:

$\mathcal{R}11$. `can_call(_,c1,_,c2) :- true.`

or to have multiple classes `c3, c4` and `c5` as acquaintance of `r`:

$\mathcal{R}12$. `can_call(r,c1,_,C) :- C= c3| C= c4| C= c5.`

5.5 A Concluding Remark: Controlled Relaxation

So far we have seen that under darwin-E, LoD can be relaxed by adding `can_call` rules to the law of the project. Such rules provide a precise definition of the exceptions to LoD that are being admitted. This scheme of relaxing a principle such as LoD can be meaningful only if there is a strict control over creation and destruction of the individual rules. Otherwise, anybody can bent the law at will simply by adding new rules or destroying existing ones. Manipulation of user-defined properties should also be strictly controlled, because one can evade the law by adding or removing the property used in existing `can_call` rules. In darwin-E, we indeed can control creation or destruction of rules, and manipulation of user-defined properties. The ability to create or destroy the `can_call` rules, for example, can easily be confined to the project managers. Similarly, the power to add or remove user-defined properties such as `status(_)` or `cluster(_)` can also be confined to the managers. More details about such control can be found in [10].

6 Related Work

Although Liberherr at al. mention that under normal circumstances, the Law of Demeter is meant to be used as a guideline [5], there are some efforts to ensure that this principle is actually satisfied by the implementation. In this section we will briefly mention the two instances that we are aware of: first, the Demeter system [3] and then CCEL [2].

LoD is used by the Demeter system in generating and transforming C/C++ code. In that sense, LoD is imposed by the Demeter system, however, it does not check user written wrappers against LoD. Although, the need for accommodating exceptions is acknolwdged [5], the Demeter methodology does not provide any flexible means like ours to do so.

An indirect approach to check C++ programs against LoD can be found in CCEL: indirect because functions that violate a certain CCEL constraint involving return values of functions also violate the law of Demeter. However, CCEL is a tool to *check* source code against certain constraints, whereas darwin-E is an environment that *imposes* the constraints. Furthermore, CCEL cannot accommodate the various exceptions we discussed.

7 Conclusion

Our treatment of the Law of Demeter in this paper constitute a demonstration of our thesis that broad structural principles can be effective if they are formally specified, strictly enforced, and relaxed whenever relaxation is in order. Many other structural principles can be treated similarly under Darwin-E.

References

[1] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[2] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A Metalanguage for C++. In *USENIX C++ Conference*, August 1992.

[3] W.L. H'ursch, L.M Seiter, and C. Xiao. In any case: Demeter. *The American Programmer*, September 1991.

[4] K. Lieberherr and I. Holland. Formulations and benifits of the law of demeter. *SIGPLAN NOTICES*, 24(3):67–78, March 1989.

[5] K. Lieberherr and I. Holland. Preventive maintenance of object-oriented software. *?*, 1992.

[6] K. Lieberherr, I. Holland, and A. Riel. Object oriented programming an objective sense of style. *SIGPLAN NOTICES :special issue on OOPSLA 88*, 23(11):323–334, November 1988.

[7] N.H. Minsky. Law-governed systems. *The IEE Software Engineering Journal*, September 1991. (This is a revision of a similarly entitled 1987 technical report).

[8] N.H. Minsky. Law-governed regularities in object systems; part 1: Principles. Technical report, Rutgers University, LCSR, December 1994. (Accepted for publication in Theory and Practice of Object Systems (TAPOS)).

[9] N.H. Minsky and P Pal. Law-governed regularities in object systems; part 2: the eiffel case. Technical Report LCSR-TR-228, Rutgers University, LCSR, December 1994. (Accepted for publication in Theory and Practice of Object Systems (TAPOS)).

[10] P. Pal and N. H. Minsky. Darwin-E : An advanced software development environment for eiffel. Technical report, Rutgers University, LCSR, 1995. (Under Preparation).