Demeter Demystified

Mitchell Wand

July 11, 1996

Abstract

My understanding of what Karl's stuff is about, what it's good for, and what its limitations are.

For the past eight years or so, Karl has been working on projects aimed at making object-oriented programming less onerous. In the process, he has developed a programming model and vocabulary that newcomers often find difficult. In this document, I will give my understanding of these projects and their significance.

This is not a scholarly survey with proper citations. It is instead impressionistic and critical. I will try throughout to be as accurate as I can, but errors in fact or interpretation are of course mine, and not Karl's. When in doubt, consult Karl's book [1].

Karl's work seems to divide into four parts:

- The Demeter system
- The Law of Demeter
- Propagation Patterns
- Adaptive Programming

1 The Demeter System

The Demeter system is a program framework for implementing the hierarchical data model in an object-oriented system. It starts with the notion of a *class graph* or *class dictionary graph*.

- In database terminology, a class graph is roughly a database schema in the hierarchical model.
- In Lisp terminology, it is roughly a `DEFSTRUCT` with type information.
- In SML terminology, it is roughly a `datatype` definition.

For example, in SML one might declare such hierarchical data by

```
datatype Exp = Ident_Exp of Ident
  | Abstraction_Exp of (Ident * Exp)
  | Application_Exp of (Exp * Exp list)
```

In SML, operations on such data would typically be specified by pattern expressions. In Pascal or C (or even in Scheme) one would generally implement such data by records containing a discriminant or tag field. Operations on this data would dispatch on the tag field.

The Demeter system provides a tool for implementing such data in an object-oriented language. It does so by creating a class `Exp` with subclasses `Ident_Exp`, `Abstraction_Exp`, and `Application_Exp`. Instead of having a single procedure `foo`, it associates a `foo` method with each subclass. (In C++, this will be a virtual method associated with the superclass `Exp`).

The Demeter system provides a set of generic tools for use with this implementation strategy, including generated parsers and unparsers, a graphical front-end, etc.

The notion of a class dictionary graph in Demeter is somewhat richer than this analogy would indicate. Class graphs in Demeter add to the SML `datatype` definition the following capabilities:

1. SML datatype definitions define *types* (such as `Exp`) and *constructors* (such as `Abstraction_Exp`). In Demeter, the corresponding entities are called `alternation classes` and `construction classes`. Also, in Demeter, the data fields have names, corresponding to instance variables. Hence in Demeter one would write something like

   ```
   Exp : Ident_Exp | Abstraction_Exp | Application_Exp .
   Ident_Exp = <identifier> Ident .
   Abstraction_Exp = "fn" <bound_var> Ident "=>" <body> Exp .
   Application_Exp = <rator> Exp <rands> "(" List(Exp) ")" .
   ```

The quoted strings specify the concrete syntax for the parser and unparser. Demeter offers a variety of input syntaxes; this is the so-called "concise" syntax.

2. Demeter also allows instance variables to be associated with alternation classes; such variables become instance variables of each of the alternatives. This gives a flavor of inheritance.

3. Demeter also allows alternation classes to have other alternation classes as alternatives. This allows the designer considerable flexibility in grouping classes. This also means that many different class graphs actually specify the same set of objects. One can decide whether two class graphs determine the same objects by transforming the class graph until it uses no inheritance (like a `datatype` declaration) and removing any dead classes. This can be done by a sequence of local transformations.

The Demeter system was originally implemented with Common Lisp Flavors as its target language. This system did not get wide usage, perhaps because it offered little value above the existing `defstruct`. The Demeter data-definition system was then ported to C++, where it got considerably more attention, perhaps because programming in C++ is so difficult that almost anything helps. We have little data on the actual usage of Demeter/C++, and still less data on how much of its usage outside Northeastern is due to the data-definition facilities and how much is due to the propagation patterns discussed below.

More recently, versions of Demeter have been developed for Stk and Perl5; versions for Java and the Booch-Rumbaugh Unified Modeling Language UML are in the works or contemplated.

Each of these implementations is a preprocessor that extracts the Demeter definitions and generates appropriate segments of code in the target language; Demeter does not attempt to process code written in the target language, such as the bodies of methods.

2 The Law of Demeter

In some object systems, instance variables and methods are assumed to be private; in others, they are assumed to be public. The Law of Demeter was an attempt to formulate a rational notion of visibility that would bridge this gap. This was claimed to increase program coherence, so that knowledge of the internals of a class would be restricted to that class and its neighbors, even if the language would permit references in other contexts.

There were various versions of the Law: strong vs. weak, run-time vs. compile-time. The Law of Demeter appears to be quite independent of the Demeter system described above.

3 Propagation Patterns

Demeter's underlying data model is that of a directed graph whose nodes are either atomic data or are labelled by their classes (the so-called *constructor classes*), and where the edges are labelled by instance variables. This graph is called the *object graph*. A class dictionary determines which such graphs are legal in any application.

One often wishes to traverse some part of an object graph, perhaps performing some operation on the nodes as they are traversed. Lieberherr observed that often one could specify such traversals by simply specifying the classes of the source and target nodes. Such a traversal specification is called a *propagation directive*. The traversal specification determines a subgraph of the object graph. This subgraph is then traversed depth-first, and actions can be performed at each node. The action to be performed at each node is called its *wrapper*, and is determined by the class of the node. Because the program executes a depth-first traversal, we can distinguish the prefix and postfix visits to a node; hence we have prefix and postfix wrappers. Actions can also be associated with edges. The entire specification, including the propagation directive and the wrappers, is called a *propagation pattern*. (I find the terminology of propagation patterns and wrappers unmotivated at best; I don't know if others find them equally discomforting).

Thus, to print out all the variables in an expression, we might write something like:

```
*operation* print_vars ()
 *traverse* *from* Exp *to* Ident_Exp
 *wrapper* Ident_Exp
  *prefix* (@ print (self.ident) @)
```

Here we have written the wrapper in pseudo-code, since Demeter does not make a commitment about the target language.

To print out all the free variables, we might write

```
*operation* print_free_vars (bound_vars)
 *traverse* *from* Exp *to* Ident_Exp
 *wrapper* Ident_Exp
  (@ if !(member (self.ident, bound_vars)
     then print (self.ident); @)
 *wrapper* Abstraction_Exp
  *prefix*
  (@ bound_vars = cons (self.bound_var, bound_vars) @)
  *suffix*
  (@ bound_vars = cdr (bound_vars) @)
```

Thus a propagation pattern is a program framework: it constructs a control structure into which the programmer can insert bits of code.

The propagation pattern tool takes a propagation pattern (traversal specification) and a class graph (schema) and produces a program that will perform the traversal on any object graph of the schema.

Full-blown propagation patterns have additional features not shown above:

- The traversal specifications may be more complex than shown above: they may include union, concatenation, etc., of paths, and the ability to specify that certain edges are not to be included.

- The operations may also include *transportation directives* that allow additional bits of state to be carried along the traversal.

The propagation pattern framework has several shortcomings:

- Its formulation in terms of paths is somewhat misleading. While propagation directives are formulated in terms of a set of paths, it is more intuitive to think of the paths as determining a subgraph for a contiguous set of nodes in the graph; then the subgraph is traversed, with the prefix and postfix actions performed at the appropriate times in the traversal.

- The semantics of a traversal specification on a graph that changes during traversal has been the subject of considerable debate within the Demeter group.
- The framework is essentially imperative. It is not so clear how to do more complicated tasks that are easily expressed recursively, such as annotating a parse tree, or constructing a copy of a parse tree with each node annotated by its set of free variables. Such operations would seem at best to require hand-simulation of the data structures involved in the recursion; the manipulation of `bound_vars` is simple enough, but more complicated examples seem to require uncomfortably much hand-coding.

4 Adaptive Programming

Because a propagation pattern usually specifies the classes of only the starting and ending nodes, the propagation pattern can be used on object graphs from a variety of class graphs (schemas). For example, the propagation patterns above would continue to work if we added a conditional expression to our language by writing

```
Exp : Ident_Exp | Abstraction_Exp | Application_Exp | Conditional_Exp .
Conditional_Exp = "if" <test_exp> Exp
                  "then" <true_exp> Exp
                  "else" <false_exp> Exp .
```

This observation is why Karl calls programming with propagation patterns *adaptive programming*: the propagation patterns adapts to changes in the schema without modification.

A propagation pattern may be thought of as polymorphic in the "type" of the graphs that it may be called upon to traverse. Karl sometimes uses the term *customizer* to denote a class graph when it used as input to a propagation pattern. (I find this terminology confusing; I would have expected the customizer to be the program that takes a propagation pattern and a class graph and produces a customized traversal program.)

A fundamental thesis of Karl's recent work is that this kind of polymorphism is useful. This is a pragmatic question, about which we have little data. While the system has been used intensively (most importantly for its own development, of course), there has been no attempt to monitor and quantify the steps by which schemata and design have evolved. Such quantitative measures would help determine the importance of this particular kind of adaptiveness.

It is likely that many evolutionary schema changes involve adding small bits of data or additional cases similar to existing ones, like adding a conditional expression to our example. With such changes, the program will continue to do *something* with the new data, but it is not clear whether what it does will be correct. For instance, consider adding a `let`-expression to our example:

```
Let_Exp = "let" <bound_var> Ident "=" <rhs> Exp "in" <body> Exp
```

Even though we continued to use the instance variable `bound_var` for the bound variable (so that the `*bypassing* bound_var` directive is still appropriate), we need to add detailed knowledge about the new construct for the program to work correctly (i.e., the variable is bound in the body but not the rhs). It seems just as likely that most schema changes involve this kind of new knowledge. It would seem that capturing such knowledge is a key to change management.

I have considered adaptiveness separately from the propagation pattern mechanism because I believe they really are separate. It is entirely possible that propagation patterns will be judged to be a worthwhile programming construct even if it turns out that the claim of adaptiveness is unjustified; and it is also possible that the study of adaptiveness or resilience in the face of change will be worthwhile even if propagation patterns are not widely adopted.

References

[1] Karl Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, MA, 1996.