Traversing Recursive Object Structures: The Functional Visitor in Demeter

Pengcheng Wu
College of Computer & Information Science
Northeastern University
Boston, MA 02115 USA
wupc@ccs.neu.edu

Shriram Krishnamurthi
Computer Science Department
Brown University
Providence, RI 02912 USA
sk@cs.brown.edu

Karl Lieberherr
College of Computer & Information Science
Northeastern University
Boston, MA 02115 USA
lieber@ccs.neu.edu

ABSTRACT

The DEMETER system provides a graph-based notation for separating the traversal of a data structure from computations over the structure. This paper presents a new programming paradigm in DEMETER that adds the power of functional composition of VISITOR methods while preserving the flexbility of traversal control.

1. INTRODUCTION

Aspect-oriented Programming (AOP) [4, 5] is a new technology for expressing the separation of crosscutting concerns in software development. Traversing an object structure and performing processing during the traversal is a common concern in software development. The VISITOR pattern [2] improves the implementation of that concern by separating the code implementing the traversal (usually the responsibility of the object structure) and the code implementing the process logic along the traversal (by defining a `Visitor` class). However, the VISITOR pattern still suffers from two limitations. The first is that it requires the object structure being visited to provide some hooks (`accept` methods) through which the `Visitor` methods can be executed along the traversal. Those hooks pollute the interfaces of the object structure since they exist solely for implementation purposes. Besides, it will be impossible to apply `Visitor` operations to a legacy object structure that wasn't implemented to support them. The second limitation is that the classic VISITOR pattern doesn't have flexible control on the traversal itself. The traversal is either implemented as hard-wired code in the object structure, which cannot make complex traversals dependent on the results of operations (defined by `Visitor` methods) on the object structure, or it is implemented in the body of `Visitor` methods, which is difficult to adapt and reuse.

The DEMETER (DEMETERJ and DJ) [7, 10, 9] work has been devoted to improving the separation of the traversal-related concerns from other concerns. In DEMETER systems, the above two limitations have been eliminated quite well. In this paper, we focus on one DEMETER programming tool DJ. DJ is a pure Java library. It supports the DEMETER methodology by providing APIs for users to implement traversals on their object structures. It avoids the first limitation of the VISITOR pattern by providing `traverse` methods in the classes of the DJ package. Concretely, it uses Java reflection [3] to traverse an object structure and to execute the `Visitor` methods during the traversal when the type of the object being traversed matches any `Visitor` method's signature. Thus the `accept` methods in the object structure side are no longer needed. The second limitation is overcome by providing a high level description language for traversal control, i.e., a *traversal strategy*, and the traversal graph itself can be computed at run time resulting in very powerful control over traversal.

However, the DEMETER system itself still has some problems, one of which is that the `Visitor` methods cannot have return values (this is also the case in the classic presentation of the VISITOR pattern), so all the computation by the `Visitor` has to be done through side-effects. That makes it inconvenient to express computations that are conveniently solvable by recursive functions, e.g., traversing a recursive data structure and returning the computation result from the structure. This paper presents our recent extension to DJ by which the `Visitor` methods can have return values and those return values can be composed in a special mechanism so that the computation along the traversal over recursive object structures can be expressed in a more natural and convenient way.

This paper is organized as follows. In Section 2, we briefly introduce the key concepts in DJ; Section 3 presents the extension introduced to DJ and shows, through an example, how the new feature helps solve the problem; Section 4 discusses related work.

2. KEY CONCEPTS IN DJ

The DEMETER system includes two adaptive Java programming tools: DEMETERJ and DJ. DEMETERJ is a static tool that injects traversal methods into classes according to the traversal strategy and combines the traversal methods with `Visitor` methods that define the computation during the traversal. DJ is a pure Java package that interprets a traversal strategy at run time and executes `Visitor` methods during traversal. The emphasis of this paper is on DJ, so here we present the key concepts used in DJ.

- *Class Graph.* The class graph is an abstraction of the class structure of the base program. Its nodes are types in the program and the edges are relationships between types, i.e., association relationships and generalization relationships. Association relationship edges have labels which are the fields' names corresponding to the relationship. In DJ, the class

graph is computed at run time by using Java's reflection mechanism [3].

- *Object Graph.* The object graph is a concrete object whose structure conforms to the class graph of the program. Usually the object graph is the object to traverse.

- *Traversal Strategy.* A traversal strategy [8] is a descriptive string that programmers use to describe the route taken by the traversal. It supports graph primitives and logic combinators, such as *from*, *to*, *via*, *bypassing*, *and* and *or*. A sample traversal strategy is *from A to B*, whose informal semantics expresses the intent to traverse from an object of type A to all the reachable objects of type B. The meaning of the strategy is defined in terms of the *first* set on nodes in object graphs of class A. An edge e outgoing from a node o of type A is in *first(o)* iff there exists an object of type *class(target(e))*[1] that can lead the traversal to reach a B-object. The formal semantics of *traversal strategy* is available in the work by Wand and Lieberherr [12].

- *Traversal Graph.* The traversal graph is a subset of the class graph, computed by applying the traversal generating algorithm to the class graph with respect to the traversal strategy. The nodes and edges in the traversal graph are the types, and relationships between types, reachable by the traversal strategy. DJ computes the traversal graph dynamically.

- *Visitor.* Programmers specify the computation to execute along the traversal in a `Visitor` class, whose methods' signatures should conform to some predefined forms described later. When the traversal reaches a point in the object graph and there is a `Visitor` method whose signature matches the traversal point, DJ executes the corresponding method. Concretely, in DJ, the signatures of `Visitor` methods should be one of the following forms (with their corresponding informal semantics):

  1. **`void before`**`(A host)` When the traversal reaches an object *o* whose type is *A*, then before traversing the parts of *o*, this **`before`** method is executed with *host* bound to *o*.
  2. **`void after`**`(A host)` When the traversal finishes traversing all the parts reachable from an object *o* whose type is *A*, this **`after`** method is executed with *host* bound to *o*.
  3. **`void before`**`(A s,String l,B t)` When the traversal is traversing an edge whose source node is the type of class *A* and whose target node is the type of class *B*, then before the traversal reaches the target node, this **`before`** method is executed with *s* bound to the source object, *t* bound to the target object and *l* bound to the label of the edge.
  4. **`void after`**`(A s,String l, B t)` When the traversal has finished traversing an object of type *B* via an edge whose source node is the type of class *A* and the target node is the type of class *B*, this **`after`** method is executed with *s* bound to the source object, *t* bound to the target object and *l* bound to the label of the edge.

[1] *target(e)* is the target node of the edge e and *class(x)* is the class of the object x.

The actual traversal begins by invoking the `traverse` method on a `ClassGraph` object with an object to traverse, a string of `traversal strategy` and a `Visitor` object as arguments. DJ can compute the traversal graph according to the class graph information and traversal strategy, and it can get the signatures and bodies of the methods defined by the `Visitor` class and traverse from an object to its parts by reflection. All the traversal and `Visitor` execution can therefore be automated.

We claim that DJ lifts the limitations of classic VISITOR pattern because:

1. It provides the traversal method as a system level API so that the `accept` methods in the base structure are unnecessary. The base class structure is even unaware of the existence of traversals or of the `Visitor`.
2. A `traversal strategy` acts as a higher level description of traversal control, which allows the programmers to pay less attention to the structural details of the base program and renders the program less sensitive to structural changes.

3. RECURSIVE PROGRAMMING STYLE

When traversing recursive object structures, it is common to compute an aggregate value for an object that is generated compositionally from the aggregate values of sub-components.

Since DJ's `Visitor` methods cannot return values, programmers implement such recursive computations in a roundabout way. They define the `Visitor` class such that different `Visitor` methods share instance variables through which the methods access the "return vales" of a traversal of sub-components, and update this variable to store the "return value" of the traversal on the object itself. This has proven to be both unnatural and error-prone for the following reasons:

- It makes it considerably more difficult for programmers to reason about their programs. The most natural solution would instead be to define recursive functions and employ them compositionally.
- The `Visitor` methods share the instance variables of the visitor by side-effects, so a variable may be updated in many methods in different traversal situations. This non-localized access makes the program more likely to generate logic errors.
- When the traversal is on a recursive object structure, to correctly access the "return values" from subcomponents, users have to manually maintain some special data structure (usually a stack) to keep track of those values.

To support our argument, we present an example to show how the DJ is usually used to solve a problem with the above properties.

3.1 An Example

Figure 1 is the UML class diagram of the example. A `Container` can contain a list of `Item`s, each of which can be either a `Simple` item or a `Container` item. `Simple` item can not contain any other items and each `Simple` item has a weight property; `Container` items can contain other items and each `Container` item has a property of capacity (for the purpose of simplicity, assume a `Container` doesn't have weight). Note that an edge from a *Collection* class to

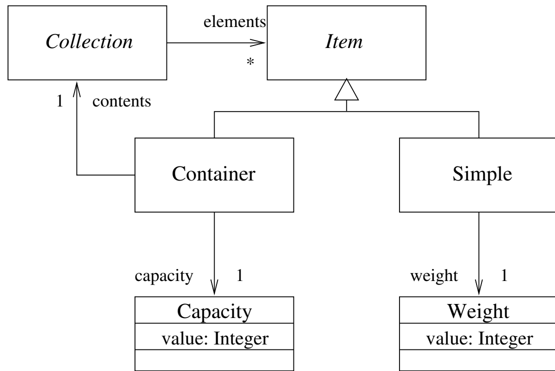

Figure 1: The UML diagram of the Container example

its contents usually has no label, while in the DJ's internal representation of *ClassGraph*, a label *elements* is always added automatically. Obviously, this structure is a recursive object structure. Assume we are supposed to ensure that the total weight of items in a container doesn't exceed the container's capacity. It is natural to think of using traversals to solve this problem.

How do we implement this checker using DJ? Currently, we would write this solution (rather inelegantly) as follows. We use a traversal strategy `''from Container to Weight''` to direct the traversal, i.e., the traversal starts from a `Container` object and tries to reach all `Weight` objects. The `Visitor` class, which specifies what computations to perform along the traversal, has the following methods:

1. **`void before`**`(Weight host)` We must add all the weights of the simple items in a container (directly or indirectly). To do so, we declare an instance variable `total` in the `Visitor` class to record the total weight so far and in this **`before`** method, we update `total` by adding the weight of the `host` to `total`.

2. **`void before`**`(Container host)` Now the `total` maintains the total weight of all the simple items having been traversed so far. What we need is the total weight of simple items held in this container. To get that, in this **`before`** method, we record the value of `total` before entering `host` by pushing `total` into a stack.

3. **`void after`**`(Container host)` When finishing traversing a container object `host`, we know the total weight of simple items contained in `host` by subtracting the value we stored in the top of the stack from `total` we have so far. Then we can check if the total weight of this container's contained items exceeds the container's capacity.

One may solve this problem in a different way in VISITOR pattern; however, the stack operations or the similar mechanisms are needed to simulate the recursive structure of the objects.

3.2 Functional Style Visitor methods

Having observed these weaknesses, we recently extend DJ to better support `Visitor` computations on recursive object structures. The main contribution of the extension is that the `Visitor` methods can now have return values, and we provide a convenient composition mechanism for users to combine the return values. The definition of the *return value* of the traversal on a node is the return value of a special `Visitor` method named **`around`**. The return type of that method is (by covariance in the return position, a subtype of) `Object`. The method has two arguments whose types are the type of the node being traversed and `Subtraversal` respectively. Class `Subtraversal` is a newly added class in DJ's API for users to explicitly invoke a subtraversal on components. The `Subtraversal` argument stores the context of the current traversal under which the subtraversal can proceed and by which the return value from the subtraversal can be accessed.

Similar to the **`before`** and **`after`** methods, an **`around`** method will be executed automatically at run time if the traversal reaches an object whose type matches the type of its first argument. An instance of `Subtraversal` will be automatically created with the traversal context and be passed to the method execution. Subtraversals from that object are invoked and the return values from them are accessed in an **`around`** method body by calling one of the following three public methods on the `Subtraversal` argument:

- `Object[] apply()` It traverses all the parts reachable from the current object in the traversal graph and returns the return values from each of the parts as an `Object` array;

- `Object apply(String edgeName)` It traverses to the non-repetition edge (in UML's terminology, the multiplicity is $0..1$) from the current object, whose label is indicated by `edgeName`, and returns the return value from the target of the edge as an `Object`;

- `Object[] applyElements()` It traverses down to the repetition edge, i.e., an edge from a *Collection* class or an array to its elements, and returns the return values from the traversal to each element as an `Object` array. The argument `edgeName` is not needed in this situation, since its label is always *elements*.

There are some situations in which we don't want to define an **`around`** on every kind of node because we wish to perform the same processing on most of them. For those nodes, users can provide a default **`around`** method in the `Visitor` class, which has the signature `Object combine(Object[] values)`. The argument is an array of return values from the traversal to each of the parts of the object being traversed. If an explicit **`around`** method is defined on a type of node, then the default `combine` method will not be executed.

Listing 1 is the source code to solve the container checking problem using the new **`around`** visitor feature. Note the default `combine` method implements the "summing up" functionality. Since there is no **`around`** method defined on `Simple` objects or `Collection` objects, the `combine` method will be applied on them automatically, thus the `apply` method call in the **`around`** method of `Container` objects will return the total weight of items in the currently visited `Container` object. The **`around`** method on `Weight` implements the base case. As is evident from the code, we no longer need the

stack operations, and the program is easier to understand, since the program is the natural recursive fashion. We have conducted other experiments with **around** visitor methods and have found that it facilitates natural recursive computations on recursive object structures.

Listing 1: ContainerChecker.java

```
// The traversal strategy is
// "from Container to Weight"
class ContainerChecker extends Visitor {
  Object combine(Object[] values) {
    int total=0;
    for(int i=0; i<values.length; i++) {
      if(values[i]!=null)
         total+= ((Integer)values[i]).intValue();
    }
    return new Integer(total);
  }
  Object around(Weight w,Subtraversal st) {
    return w.get_value();
  }
  Object around(Container c,Subtraversal st) {
    Integer total = (Integer)st.apply("contents");
    if(total.intValue() > c.get_capacity().
      get_value().intValue())
      System.out.println("An Overloaded Container");
    return total;
  }
}
```

3.3 Greater Dynamic Traversal Control

In addition to adding support for a recursive programming style, we now have greater dynamic control over traversals in DJ. This was not supported well in previous DJ versions.

With the introduction of the `Subtraversal` class, we can now decide whether we want the traversal to go further along an edge depending on the results of run-time values by choosing to invoke or not invoke those `apply` methods. We can even traverse the same edge (and the corresponding target objects) multiple times by calling an `apply` method multiple times with the same edge label as the argument. In some situations, it is also important that the traversal *order* of the part objects of an object be programmable. We now can specify the order by choosing the order in which we call the `apply` methods. Of course, any path we choose to traverse at run time must have been in the set of paths in the traversal graph computed from the traversal strategy.

4. RELATED WORK

The Demeter work on traversals is a generic tool for solving problems related to crosscutting concerns that can be abstracted as graphs. In one application, the graph is the dynamic call graph of the execution of an object graph traversal. The crosscutting concern selects a subset of the nodes and edges in the object graph traversal. This kind of selection can also be achieved with a general purpose aspect-oriented language where we can select general dynamic join points, not just the ones corresponding to object graph traversal. Therefore we compare now to ASPECTJ [1] as the most prominent aspect-oriented language.

4.1 Related work in AspectJ

The support in DJ for **around** methods with a non-void return value enhances the expressiveness of the DEMETER system and provides for a very symmetric mapping between the Join Point Model (JPM) of ASPECTJ [1] and that of DJ.

4.1.1 Symmetric Mapping to AspectJ's JPM

The JPM of an AOP language specifies how the implementations of crosscutting concerns are integrated (or woven) together to form a functional system. Generally speaking, when we design a JPM, three questions should be answered about the model:

1. What are the possible join points to which the implementations of other concerns can be integrated?
2. How to select a subset of those join points where integration should happen?
3. How to specify what needs to be integrated at the points specified by item 2.

As a general purpose AOP language, ASPECTJ has join points in dynamic call graph (for 1) and provides a pointcut definition language (for 2) and the advice mechanism (for 3).

DJ is a specialized AOP system, which supports modular implementation of traversal-related concerns, by providing a traversal strategy description language that defines a traversal whose node and edge visits we care about (for 1) and a Visitor mechanism (for 2 and 3). The visitor method signatures select the join points that we want to enhance (for 2) and the visitor method bodies provide the enhancement code (for 3). For details see [9] (DJ class `Visitor`).

Table 1 lists the answers to the three questions for ASPECTJ and DJ, respectively.

	ASPECTJ	DJ
Join Points	The points in the dynamic call graph of the program	The nodes/edges in the object structure specified by a traversal strategy
Where	The points in call graph selected by pointcuts	The nodes/edges selected by visitor signatures
What	before/around/after advice	before/around/after Visitor method bodies

Table 1: The Join Point Model of ASPECTJ and DJ

A join point model (besides having the three properties mentioned above) also exposes context information about the join points. The JPM of DJ operates on object graph slices (object graphs selected by a traversal specification) while the JPM of ASPECTJ operates on dynamic call graphs. Table 2 shows symmetric mappings between the two tools. However, it is important to note that with DJ (which can be used simultaneously with ASPECTJ) it is much more convenient to express traversal-related concerns than with ASPECTJ alone.

4.2 Other Traversal-related Work

In his paper about visitor combination and traversal control [11], Joost Visser presents a novel way to add traversal control to the classic VISITOR pattern. The traversal control is on the visitor side, that is, he defines some generic Visitor combinators which can provide common traversal orders, such as bottom up, top down, etc., and a Visitor object can be passed to those combinators to achieve the goal of applying the visitor to a data structure in a defined order. Unlike the Demeter system, this work is using a standard object-oriented framework approach: the `accept` methods on the base class structure are still needed.

ASPECTJ	DJ
p(A a):target(a)&&call(* *(..))	Visitor method signature (A a)
p(A a,B b):this(A)&&target(b) &&call(* *(..))	Visitor method signature (A a,String l,B b)
around advice having return value	around Visitor method having return value
In around advice, one can decide whether to proceed by calling proceed()	In around methods, one can decide whether to continue one branch of the traversal by calling apply() methods
Reflection: thisJoinPoint	Reason about traversal: Subtraversal

Table 2: Parallels between ASPECTJ and DJ

Strategic Programming [6] is a generic programming idiom for processing compound data by separating basic data-processing computations from traversal schemes. We view it as a more general model of [11]. The power of Strategic Programming lies in the power of the combinators of strategies, i.e., users can compose complex traversal strategies by applying combinators to simpler strategies, while the strength of the Demeter work is the efficient computation of traversals. It is not clear if Strategic Programming supports the functional composition capability presented in this paper.

5. CONCLUSION

We propose a better approach to aspect-oriented programming of traversal-related concerns on recursive object structures. We introduce **`around`** methods with non-void return values and a combination mechanism into DJ visitors and add a selective traversal continuation construct in **`around`** methods.

The paper makes a contribution to the domain of concern-specific aspect languages. Using the extended DJ tool presented in this paper it is more convenient to address traversal-related concerns than doing it directly in the general purpose aspect language ASPECTJ.

Acknowledgements

We are grateful to Mitchell Wand for pointing out the inconvenience of expressing certain computations on recursive object structures in the DEMETER system. Many thanks to Doug Orleans for very helpful discussions about the functional Visitor in DJ.

6. REFERENCES

[1] X. P. AspectJ Team. AspectJ home page. http://aspectj.org. Continuously updated.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[3] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison-Wesley, 2 edition, 2000.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. Knudsen, editor, *European Conference on Object-Oriented Programming*, pages 327–355, Budapest, 2001. Springer Verlag.

[5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[6] R. Lämmel, E. Visser, and J. Visser. Strategic Programming Meets Adaptive Programming. In *Proc. of the 2nd International Conference on Aspect Oriented Software Development*. ACM Press, 2003. 10 p.; To appear.

[7] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[8] K. J. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997. http://www.ccs.neu.edu/research/demeter/AP-Library/.

[9] D. Orleans and K. Lieberherr. Demeter API home page. http://www.ccs.neu.edu/research/demeter/software/docs/api. Continuously updated.

[10] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of C rosscutting Concerns* , pages 73–80, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[11] J. Visser. Visitor combination and traversal control. In *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices*, pages 270–282. ACM, October 2001.

[12] M. Wand and K. Lieberherr. Traversal semantics in object graphs. Technical Report NU-CCS-2001-05, Northeastern University, May 2001.