Recursive Adaptive Computations Using Perobject Visitors

Ahmed Abdelmeged
Northeastern University
mohsen@ccs.neu.edu

Karl Lieberherr
Northeastern University
lieber@ccs.neu.edu

Abstract

Adaptive Programming allows developers to write structure-shy programs. However, in Adaptive Programming, recursive computations are known to require a good deal of boiler plate code to express. This paper describes *Perobject Visitors*; a programming construct that allows developers to write recursive adaptive computations at a higher level of abstracton. This paper also describes a prototype implementation for perobject visitors.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Modules and interfaces, Object-oriented design methods; D.3.3 [*Language Constructs and Features*]: Patterns, Recursion

General Terms Design, languages

Keywords aspect-oriented programming, temporary intertype declaration, visitor pattern

1. Introduction

The well known visitor design pattern [4] provides a mechanism for attaching new functionality to existing class structures. While doing so, it achieves a clean separation of the navigational code in all *accept* methods from the computational code in all *visit* methods.

Adaptive programming (AP) [6] takes the visitor design pattern one step further; in AP, developers write a traversal specification in a high level domain specific language. The traversal specification then gets compiled against the class structure of the application to generate all necessary *accept* methods.

In the visitor pattern, the accept methods call the *visit* methods. Therefore, a *visit* method does not invoke its successors' *visit* method directly. Instead, it returns to its calling *accept* method which, in turn, invokes the next *visit* method. Therefore, a *visit* method cannot postpone a pending operation until the successor *visit* method finishes. As a result, writing a set of *visit* method to implement a non-tail-recursive computation becomes hard.

AP provides the developer with two specialized types of *visit* methods: *before* and *after*. Developers can use a *before* method exactly as they use an old-fashioned *visit* method. Every *before* method has a corresponding *after* method that can be used to express the pending operations. At runtime, an invocation of a *visit* method is associated with an object that is passed as a parameter to the method. *Visit* methods are typically bundled together in a visitor class.

A shortcoming of AP is that dual invocations of *visit* methods have separate environments. A typical work around is to use the visitor class as a blackboard for sharing environments between *visit* methods. AP does not provide any special support for sharing environments even between dual invocations of *visit* methods.

In case of recursive object structures, it becomes necessary to organize the blackboard using a stack. Using the stack adds two unnecessary complications to the code: first, code for initializing and managing the stack. Second, variables must be accessed and mutated while they are on the stack.

This paper presents perobject visitors. Perobject visitors are normal visitors that uses perobject variables. A perobject variable is a boiler-plate-free programming construct for sharing environments between a set of invocations of *visit* methods. Perobject visitors are not tied to AP and can be easily adapted to other situations, for example, to the general visitor pattern.

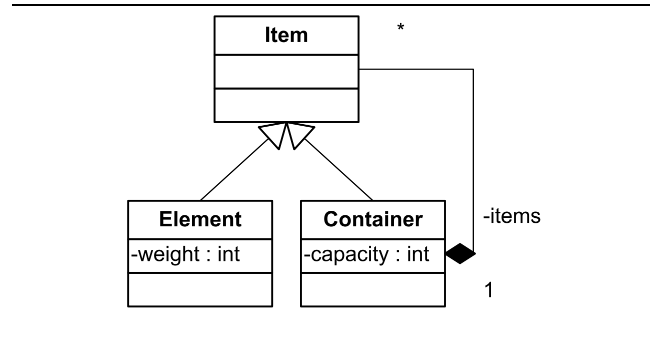

Figure 1. Containers UML class diagram

1.1 Motivating Example

In the capacity checking problem [3], containers are composites that contain elements as well as other containers. Elements are primitives that have weight. Every container has a capacity. Fig. 1 shows the UML class diagram of containers. The weight of a container is the sum of the weights of all elements that are inside it either directly or indirectly. A container whose weight exceeds its capacity is said to be violating the capacity condition. Given a certain configuration of containers, it is desired to count the total number of violations to the capacity condition.

2. Perobject Variables

The code in Fig. 2 shows a solution to the container checking problem using visitors. There are two points about this code that are worth making: First, a new integer is allocated and pushed on top of the stack whenever a container object is encountered during the traversal. This is done by the *before* method associated with that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Oopsla '07 October 21-25, 2007, Montreal.
Copyright © 2007 ACM [to be supplied]. . . $5.00

container object. There is no other place where anything is pushed on top of the stack. Therefore, every element in the stack has an associated container object. Whenever an *after* method is executed for a certain container object, the integer on top of the stack at that time is the integer associated with its container object which has been pushed by its corresponding *before* method.

Second, whenever an *after* method is executed for a container object *c*. The second integer on top of the stack is associated with the container object that encloses *c*.

```
public class CheckerVisitor extends Visitor{
    int totalViolations = 0;
    Stack < MutableInteger > myWeight =
      new Stack < MutableInteger > ();
    public CheckerVisitor()
      { myWeight.push(new MutableInteger(0)); }
    public MutableInteger currentMyWeight()
      { return myWeight.peek(); }
    public MutableInteger previousMyWeight()
      { return myWeight.elementAt(myWeight.size()-2); }
    public void before(Container c)
      { myWeight.push(new MutableInteger(0)); }
    public void after(Container c){
      previousMyWeight().setI(previousMyWeight().getI()
                              + currentMyWeight().getI());
      if (c.capacity < currentMyWeight().getI())
          totalViolations ++;
      myWeight.pop();
    }
    public void before(Element e){
      currentMyWeight().setI(currentMyWeight().getI()
                             + e.weight);
    }
}
```

Figure 2. Solving the container checking problem using DJ visitors

Perobject visitors simplify this code by using a perobject variable myWeight (Fig. 3) whose scope is a set of classes. The scope of myWeight is the entire traversal of Container while the extent of a value of myWeight either reaches down to an Element-object or another Container-object without traversing into that Container-object. Instead of using a stack, the developer declares a normal integer field in the visitor class and annotates it to be a perobject variable. In this case, a new integer field with the same name will be added to the every container object. Since, the same container object is passed as a parameter to the two corresponding *visit* methods associated with it, the newly added field becomes automatically shared between the two dual *visit* methods.

Moreover, during the execution of any *visit* method, the integer field that is declared in the visitor class and has the perobject annotation becomes an alias of the newly added field to the innermost enclosing container. We call this the aliasing invariant. The code shown in Fig. 3 uses perobject variables to implement exactly the same thing as the code in Fig. 2.

A perobject variable defined at class C is accessible in all *before*/*after* methods of classes that are reachable from C without visiting C again or by visiting C and stopping at C. If traversal strategies are used to define traversals (this is a special application of the general perobject visitor pattern) then we can be more precise about the scope and extent of a perobject variable.

A traversal strategy is a general graph sg with source s and target t [5]. When applied to a class graph cg, a traversal graph tg (basically the cross product of sg and cg) defines the scope of the strategy. We assume the special case where the perobject variable is at the source of the strategy: The scope of the perobject variable is the set of all classes in the traversal graph for sg and cg. The extent of a specific value of the perobject variable is the union of the following set of objects:

```
public class CheckerV isitor extends Visitor{
    int totalViolations = 0;
    @PerObject(
        className = "Container";
        initializer = "0"
    ) int myWeight;
    public void before(Element e){
        myWeight = myWeight + e.weight;
    }
    public void after(Container c){
        myWeight = myWeight + c.myWeight;
        boolean vio = c.capacity < c.myWeight;
        if (vio){
            totalViolations++;
        }
    }
}
```

Figure 3. Solving the container checking problem using perobject visitors

- The objects selected by the strategy $sg2$ which is like sg but "$bypassing\ s$" has been added to every edge of sg. $sg2$ visits all objects that are not contained in another s object.
- The objects selected by strategy sg that are also selected by the strategy from "s *to-stop* s". This selects all top-level s-objects.

3. Implementation

As a prototype, we developed an *annotation processor* for Java source files containing perobject annotations[2]. We build our implementation on top of DJ which is a Java library for AP [7]. The impelemented annotation processor generates an AspectJ [1] file that introduces necessary fields, initializes these fields by advising the appropriate constructor, and mantains the aliasing invariant mentioned in section 2.

References

[1] The AspectJ Project. www.eclipse.org/aspectj/.

[2] Perobject Visitors. http://www.ccs.neu.edu/home/mohsen/perobject/, 2007.

[3] Bryan Chadwick and Therapon Skotiniotis and Karl Lieberherr. Functional Visitors Revisited. Technical Report NU-CCIS-06-03, Northeastern University, Boston, May 2006.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] Karl Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[6] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, May 1994.

[7] Doug Orleans and Karl J. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.