# 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 inter-type 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) [8] 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.

The rest of this paper is organized as follows: section 1.1 presents a motivating example that shall be used also as a running example. Section 2 presents perobject variables. In section 3 we describe our implementation of perobject variables. Section 4 discusses some of the related and future work. Section 5 concludes this paper.

### 1.1 Motivating Example

In the capacity checking problem [2], 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 contain-
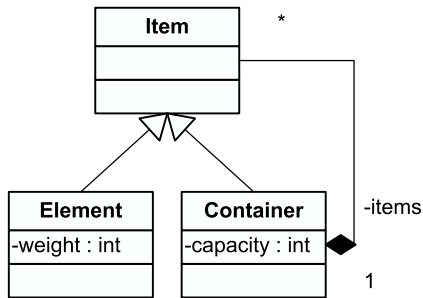
**Figure 1.** Containers UML class diagram

ers. 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 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 $c$.

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

```java
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

aliasing invariant. The code shown in Fig. 3 uses perobject variables to implement exactly the same thing as the code in Fig. 2.

```java
public class CheckerVisitor 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 visitors

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 $t$ [7]. 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:

- 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[1]. We build our implementation on top of DJ which is a Java library for AP [9]. Fig. 4 shows the perobject annotation type. Perobject annotations annotate local variables. A perobject annotation has two elements: *className* and *initializer*. The className element specifies a class that will be extended to store a copy of the annotated local variable. The initializer specifies the initial value of the newly introduced store.

```
public @interface PerObject{
    String className();
    String initializer();
}
```

**Figure 4.** Perobject annotation type

The annotation processor generates an aspect that handles three tasks:

1. Introduce necessary fields to the classes specified in annotations.

2. Initialize the introduced fields.

3. Maintain the aliasing invariant mentioned in section 2.

The first task is achieved via AspectJ intertype declarations. The second task is achieved by advising the constructor. Fig. 5 shows a snippet of the AspecJ code that is responsible for maintaining the aliasing invariant in the code shown in Fig. 3. The two situations where the aliasing invariant breaks are when the visitor nests down into the structure and when it backs up. The two events that signal these two situations are: the end of a *before* method, and the beginning of an *after* method. The two pointcuts in Fig. 5 trap these two events. To illustrate how the aliasing invariant is maintained at these two situations consider the the following example: suppose that there are three nested container objects $C1$, $C2$, $C3$, where $C2$ is nested inside $C1$ and $C3$ is nested inside $C2$. Suppose that there is a visitor class $V$ that has a perobject variable $p$ at container objects. Suppose that the before method invocation at $C2$ has just ended and it's time to move on to $C3$. At this time, $V.p$ holds the latest value of $C1.p$. We swap $V.p$ and $C2.p$. By doing so, we hit two birds: first, $V.p$ has the value of $C2.p$ and that is needed to keep the aliasing invariant when moving on to $C3$. Second, $C2.p$ saves the latest value of $C1.p$ till the visitor comes back to $C2.p$. At this point, $V.p$ will be holding the latest value of $C2.p$ and $C2.p$ will be holding the latest value of $C1.p$. Now, we do another swap and the aliasing invariant is met.

```
pointcut beforeContainer(CheckerVisitor v, Container c):
        target(v) && args(c) && within(CheckerVisitor)
        && execution(public void before(Container));

after(CheckerVisitor v, Container c):
 beforeContainer(v,c){
        int tmp = c.myWeight;
        c.myWeight = v.myWeight;
        v.myWeight = tmp;
}

pointcut afterContainer(CheckerVisitor v, Container c):
        target(v) && args(c) && within(CheckerVisitor)
        && execution(public void after(Container));

before(CheckerVisitor v, container.Container c):
 afterContainer(v,c){
        int tmp = c.myWeight;
        c.myWeight = v.myWeight;
        v.myWeight = tmp;
}
```

**Figure 5.** AspectJ code for keeping the perobject invariane

## 4. Related and Future Work

Ovlinger and Wand [10] propose a domain specific language as a means to specify recursive traversals of object structures used with the visitor pattern. The domain specific language provides traversal flexibility at a higher level than hand-coded traversals, but is not robust with respect to data structure changes, unlike AP. With perobject variables we achieve an important goal: We can express complex recursive traversals in a structure-shy way which was not possible before.

The idea of environmental acquisition [5] is related to perobject variables. According to Lorenz[3], environmental acquisition was invented in the context of the visitor pattern. While with environmental acquisition information is acquired through the (reversed) containment structure using explicit declarations, perobject variables are generously broadcast through the containment structure specified by some traversal.

Perobject visitors are a special kind of aspect; perobject visitors define temporary intertype declarations (or introductions) [6], extending the class hierarchy for the duration of a traversal. While AspectJ has intertype declarations that are all permanent, the programmer can create aspects with pointcuts that allow access only during a traversal.

As for normal visitors, the visitor method signatures define pointcuts on the traversal and the visitor method bodies define advice. Perobject visitors also provide additional pointcuts and advice to maintain access to the enclosing object of a perobject variable.

Perobject visitors were inspired by functional visitors [2]. Functional visitors also support structure-shy expression of

complex recursive traversals as well as powerful composition of visitors; but the the communication between visitor methods introduced by perobject variables makes many programming tasks easier.

Regarding future work we have at least two items on the agenda:

- We want to study composition of perobject visitors. It appears that normal inheritance is quite powerful in this respect, although it is not perfect. For example, we have decomposed the tangled code in Figure 3 in a sequence of 4 inheriting visitors: CounterAccumulator inherits from Updater which inherits from VioVisitor which inherits from WeightVisitor. WeightVisitor is responsible for summing the weights, VioVisitor for computing whether we have a violation, Updater is responsible for invoking an accumulator if there is a violation and finally CounterAccumulator is a specific accumulator that counts. The VioVisitor and the Updater communicate through a perobject variable. Such a composition has the advantage that we can easily replace parts. For example, instead of counting the violations, we may want to collect the violating containers into a list. This is achieved by replacing the CounterAccumulator by a ListAccumulator.

- Our current implementation builds on a heavily reflective implementation [9]. For example, fig. 6 shows the entry point for the CheckerVisitor class from fig. 3: Currently,

```
ClassGraph cg = new ClassGraph(true, false);
cg.traverse(myContainer,
            "from Container to Element",
            new CheckerVisitor());
```

**Figure 6.** Entry point for CheckerVisitor

the traverse method is executed using reflection even when the necessary information is known statically. We would like to analyze the way the visitor uses both the traversal and perobject variables and generate more efficient code. For example, when a perobject variable is always used for the current object and not for the enclosing object, there is no need to activate the before/after advice that maintains the aliasing invariant.

## 5.  Conclusions

In this paper we discussed the problem of expressing recursive adaptive computations. We presented perobject visitors as a pattern for writing boiler-plate-free recursive adaptive computations. We also presented a prototype implementation based on Java annotations and AspectJ code generation. We have applied perobject visitors to both query tasks (like the container example we used in this paper) and translation tasks (like reducing a CNF formula in a SAT solver). Perobject variables are a useful addition to the general Visitor design pattern as well as to the more specialized Adaptive Programming.

## References

[1] Ahmed Abdelmeged. Perobject Visitors. `http://www.ccs.neu.edu/home/mohsen/perobject/`, 2007.

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

[3] David H. Lorenz. private communication, 2007.

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

[5] Joseph Gil and David H. Lorenz. Environmental Acquisition–A new inheritance-like abstraction mechanism. In *Proceedings of the 11$^{th}$ Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214–231, San Jose, California, October 6-10 1996. OOPSLA'96, Acm SIGPLAN Notices 31(10) October 1996.

[6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jorgen Knudsen, editor, *European Conference on Object-Oriented Programming*, Budapest, 2001. Springer Verlag.

[7] 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.

[8] 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.

[9] 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.

[10] Johan Ovlinger and Mitchell Wand. A language for specifying recursive traversals of object structures. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 70–81, 1999.