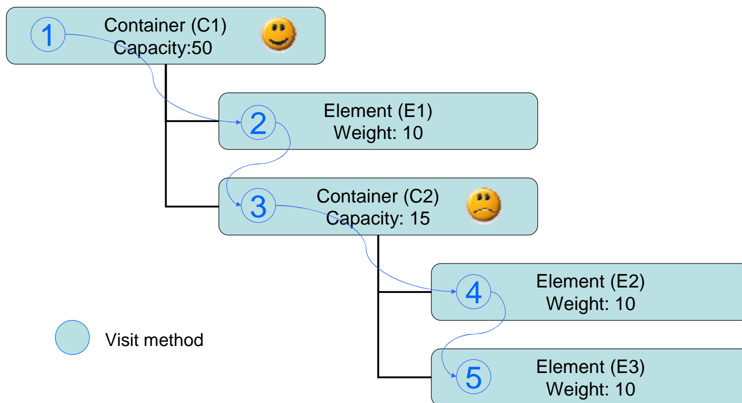
Recursive Adaptive Computations Using Interposition Visitors

Ahmed Abdelmeged and Karl Lieberherr
College of Computer and Information Science
Northeastern University
{mohsen,lieber}@ccs.neu.edu

Problem with visitor pattern[1]

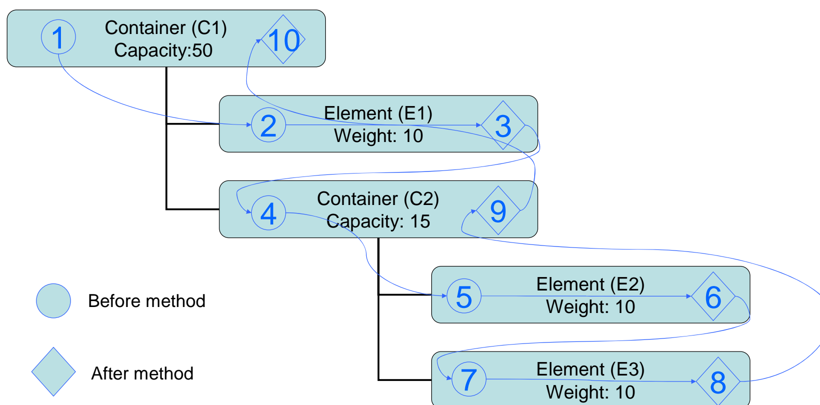
How many containers carry more than their capacity?[2]



Hint: Capacity checking needs to be done only "after" summing the weights of all children.

Cannot implement using the shown sequence of visit method invocations

Problem with Adaptive Programming[3]



Environment of Before C1 is not accessible in:
- After C1.
- Before E1, After E2, Before C2, After C2.

Anatomy of a before method

```
Class Visitor{
      int foo;
      …
      void before(Container c){
            c.foo = foo;
            …
      }
}
```

Visitor fields are accessible here

Visited object fields are accessible here

Interposition variables

A perobject variable called “MyWeight” introduced at class Container is:

1. Visible in before/After Container through the parameter.
2. Aliased in the visitor while visiting children of a container.

Visit method	MyWeight @ C1	MyWeight @ C2
1 Before C1	●	
2 Before E1	○	
3 After E1	○	
4 Before C2	○	●
5 Before E2		○
6 After E2		○
7 Before E3		○
8 After E3		○
9 After C2	○	●
10 After C1	●	

● Perobject variable seen as a part of the visited object.

○ Perobject variable aliased as visitor field.

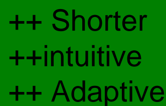
Perobject variables Vs. others

```
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++;
        }
    }
}
```

```
public int[] check(Item o){
   int myWeight = 0;
   int violations = 0;
   if(o instanceof Container){
      Container c = (Container) o;
      for (Iterator iterator = c.items.iterator();
      iterator.hasNext();) {
         Item item = (Item) iterator.next();
         int [] ret = check(item);
         myWeight+=ret[0];
         violations+=ret[1];
      }
      if(myWeight>c.capacity) violations++;
   }else{
      Element e = (Element) o;
      myWeight = e.weight;
      violations = 0;
   }
   return new int[]{myWeight,violations};
}
```

```
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);
    }
}

Public class MutableInteger{
…
}
```



Implementation[4]

An annotation processor generates an aspect that introduces necessary variables, initialize them, and implement the aliasing.

```
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;
 }
```

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

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

[3] 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.

[4] Perobject Visitors. http://www.ccs.neu.edu/home/mohsen/perobject/.