Object-Oriented Design — **Lecture 18**
CSU 370 Fall 2007 (Pucella) — **Friday, Nov 16, 2007**

The Transformer Design Pattern

Last time, we looked in detail at the Visitor design pattern. Recall that the idea there was to figure out a way to traverse a structured object, and "do something" at every element of the structured object that we traverse. We did this by decoupling the traversal from the action to be performed at every element. The idea being that we only need to implement the traversal code once, in the classes representing the objects to be traversed, and we then only need to parameterized the traversal by an object that captured what actions should be performed at each element.

One limitation we saw last time was that the traversal did not return any value. Indeed, the `accept` method implementing the traversal has a `void` return type. This means that the only thing really we can during a traversal is perform a side-effecting operation. For example, we saw that this works perfectly well for printing out something for every element we traverse. If we actually need to return a value from the traversal, however, things were a bit uglier. The one example we had was returning the minimum value in a binary tree. We did so by defining an instance variable in the visitor that would hold the minimal value we had seen until now in the traversal, and updating that instance variable everytime we encountered a node with a smaller value. Once the traversal was done, we could access the instance variable through the method `getMin` of the visitor and extract the minimum value that way.

The purpose of this lecture is to show how the Visitor pattern we saw last time could be generalized to return a value from the traversal.

We are going to do this by extending the Visitor interface so that instead of simply visiting each element, we also return a value from the visit. Moreover, the visit will not only take the current element as input, but we will also supply the value returned from the traversal of the subelements. The resulting interface we will call a *Transformer* interface.

Let's look at the binary trees example from last lecture again. The corresponding transformer interface is given by:

```
public interface TreeTransformer<U> {
    public U transform (EmptyTree t);
    public U transform (NodeTree t, U u1, U u2);
}
```

The interface is parameterized by the return type of the traversal.

Let us see what the traversal looks like for trees. In the Visitor design pattern, we implemented an `accept` method for every element that could be traversed. Here, we implement a `reduce` method for every element that could be traversed. Because reductions may return different types, we will implement different `reduce` methods working at different types.

The first `reduce` method we will implement will accept a tree transformer returning integers, and itself return an integer.

```
public abstract Tree {

  ...

  public abstract Integer reduce (TreeTransformer<Integer> tr);
}


public EmptyTree extends Tree {

  ...

  public Integer reduce (TreeTransformer<Integer> tr) {
      return tr.transform (this);
  }
}


public NodeTree extends Tree {

  ...

  public Integer reduce (TreeTransformer<Integer> tr) {
      Integer lval = left.reduce(tr);
      Integer rval = right.reduce(tr);
      return tr.transform(this,lval,rval);
  }
}
```

We see, in the above traversal code, that when we invoke the transform method on a node, we pass in the result of traversing the left and right subtrees.

Let's look at some transformations. First, an easy integer-valued transformation is one that computes the height of a tree.

```
public class HeightTransformer implements TreeTransformer<Integer> {
    public HeightTransformer () { }

    public Integer transform (EmptyTree t) {
      return 0;
    }

    public Integer transform (NodeTree t, Integer l, Integer r) {
      return 1 + Math.max(l,r);
    }
  }
```

Study the above code, and make sure you understand how it works. Here is how it can be used.

```
  Tree e = Tree.empty ();
  Tree t = Tree.node (99, Tree.node (66, e,e),
  Tree.node (33, e,e));
  int height = t.reduce(new HeightTransformer());
```

A related transformation returns the total numbers of nodes in a tree:

```
  public class SizeTransformer implements TreeTransformer<Integer> {

    public SizeTransformer () { }

    public Integer transform (EmptyTree t) {
      return 0;
    }

    public Integer transform (NodeTree t, Integer l, Integer r) {
      return 1 + l + r;
    }
  }
```

Using this transformer is as easy as using the height transformer:

```
  int size = t.reduce(new SizeTransformer());
```

Let's now give a transformer for computing the minimum value in a tree.

```
  public class MinTransformer implements TreeTransformer<Integer> {
    public MinTransformer () { }

    public Integer transform (EmptyTree t) {
      return null;
    }

    public Integer transform (NodeTree t, Integer l, Integer r) {
      int curr = t.root();
      int l2 = (l==null? curr : l);
      int r2 = (r==null? curr : r);
      return Math.min (curr,Math.min(l2,r2));
    }
  }
```

Compare this code to the visitor-based implementation of extracting the minimum we saw last time. This one is cleaner, in the sense that it does not rely on side effects to perform its computation. As we already argued many times, this leads to code that is much easier to reason about. There is still some ugliness having to do with the use of `null`, though, which is somewhat unavoidable—after all, taking the minimum value of an empty tree is impossible, so we cannot expect a clean solution.

What about transformations that return other values than integers? A general class of transformation is that returns a new tree after the traversal. The structure of the traversal is as before, except that we return a value of type `Tree`, and moreover, during the transformation of a node, we pass into the `transform` method not only the current node but the `Tree` result of transforming the left and right subtree—these can be used to reconstruct a new tree.

First off, here is the traversal code.

```
  public abstract Tree {

    ...

    public abstract Tree reduce (TreeTransformer<Tree> tr);
  }


  public EmptyTree extends Tree {

    ...

    public Tree reduce (TreeTransformer<Tree> tr) {
        return tr.transform (this);
    }
  }


  public NodeTree extends Tree {

    ...

    public Tree reduce (TreeTransformer<Tree> tr) {
        Tree lval = left.reduce(tr);
        Tree rval = right.reduce(tr);
        return tr.transform(this,lval,rval);
    }
  }
```

Note that we are using Java overloading to use the same method name `reduce` to the transformation traversal, where the method to use depends on the type of the transformer passed as an argument.

So what kind of transformers can we implement? An easy one is one that creates a new tree in which every node is double the value in the original tree. Here is the transformer:

```
  public class DoubleTransformer implements TreeTransformer<Tree> {

    public DoubleTransformer () {  }

    public Tree transform (EmptyTree t) {
      return t;
    }

    public Tree transform (NodeTree t, Tree l, Tree r) {
      return Tree.node (t.root() * 2, l, r);
    }
  }
```

Again, make sure you understand the above code. Using the sample tree `t` given earlier, here is a sample use:

```
  Tree doubled = t.reduce(new DoubleTransformer());
```

A slightly more involved transformation that generalizes the above idea is to transform a tree by creating a new tree in which every node gets its original value plus some constant specified at the time the transformer is created.

```
public class AddTransformer implements TreeTransformer<Tree> {

  private int toAdd;

  public AddTransformer (int i) {
    this.toAdd = i;
  }

  public Tree transform (EmptyTree t) {
    return t;
  }

  public Tree transform (NodeTree t, Tree l, Tree r) {
    return Tree.node (t.root() + toAdd, l, r);
  }
}
```

To use, you specify the value to add to every node at creation time:

```
Tree added = t.reduce(new AddTransformer(10));
```

Transformations can also return a new tree that has a different structure than the one being traversed. For example, the following transformer traverses a tree and reconstruct a new tree that look like the original except that every empty subtree in the original is now replaced by a node with a value give at construction time and no subtrees.

```
public class EmptyTransformer implements TreeTransformer<Tree> {

  private int newNodes;

  public EmptyTransformer (int i) {
    this.newNodes = i;
  }

  public Tree transform (EmptyTree t) {
    return Tree.node(this.newNode, Node.empty(), Node.empty());
  }

  public Tree transform (NodeTree t, Tree l, Tree r) {
    return Tree.node (t.root(), l, r);
  }
}
```

Using this is similar to using `AddTransformer`:

```
  Tree extended = t.reduce(new EmptyTransformer(77));
```

This creates a new tree `extended` that looks just like `t` except every empty subtree is replaced by a tree with a single node with value 77.

Another easy kind of transformation is a "deep swap":

```
  public class DeepSwapTransformer implements TreeTransformer<Tree> {

    public DeepSwapTransformer () {}

    public Tree transform (EmptyTree t) {
      return t;
    }

    public Tree transform (NodeTree t, Tree l, Tree r) {
      return Tree.node (t.root(), r, l);
    }
  }
```

You can draw a few pictures to see how this transformer works.

Advanced Topic 1: A General Map Transformer

Note the similarity between the `DoubleTransformer` and `AddTransformer` classes. They both apply a transformation function to the value of the node, before reconstructing a tree that has the same shape as the original. We can therefore consider these two transformations as transformations that copy a tree but change the value at the nodes in a way specified by a node transformation function. As you may recall from previous courses, this is called a `map` operation: we map a given function over a data structure, creating a copy of the data structure in which every value is the result of applying the function to a value in the original structure. For instance, in Scheme, mapping the function `(lambda (x) (+ (* 3 x) 1))` over the list `(3 5 7 9)` yields the new list `(10 16 22 28)`.

We can easily generalize both `AddTransformer` and `DoubleTransform` into a `MapTransformer` that transforms a tree into a new tree obtained by applying a given function to every value stored in the nodes of the original tree. In order to do this, we need to figure out a way to pass a function to the map transformer. Java does not let you pass functions (or methods) around as arguments like Scheme does, but what we can do is wrap a method into a class, and pass an instance of that class as an argument. Presto, first-class functions in Java. With that in mind, here is a possible implementation of a map transformer:

```
public class MapTransformer implements TreeTransformer<Tree> {

  public static interface Function {
    public int aply (int arg);
  }

  private Function nodeTrans;

  public MapTransformer (Function n) {
    this.nodeTrans = n;
  }

  public Tree transform (EmptyTree t) {
    return t;
  }

  public Tree transform (NodeTree t, Tree l, Tree r) {
    return Tree.node (nodeTrans.apply(t.root()), l, r);
  }
}
```

We're using a nested interface here, called `Function`, which wraps a given operation that we want to apply to every value in the tree. To use such a transformer, we need to create an instance of `MapTransformer.Function` to represent the function to apply, and construct a `MapTransformer` with that instance as an argument. Here is the Java version of mapping `(lambda (x) (+ (* 3 x) 1))` over a tree. First, we define the actual function to apply at every node:

```
public class Collatz implements MapTransformer.Function {
  public Collatz ();
  public int apply (int arg) {
    return 3 * arg + 1;
  }
}
```

and then we can simply invoke the transformer:

```
Tree result = t.reduce(new MapTransformer(new Collatz()));
```

Advanced Topic 2: Even More Genericity

We can make the above Transformer design pattern even more generic by noting that our trees themselves are not as generic as they could be. In particular, we could easily make the class `Tree` generic over the type of values stored in the trees. Let's do that:

```
public abstract class Tree<T> {

  public static <T> Tree<T> empty () {
    return new EmptyTree<T> ();
  }

  public static <T> Tree<T> node (T val, Tree<T> l, Tree<T> r) {
    return new NodeTree<T> (val,l,r);
  }

  public abstract boolean isEmpty ();
  public abstract T root ();
  public abstract Tree<T> left ();
  public abstract Tree<T> right ();
}


class EmptyTree<T> extends Tree<T> {

  public EmptyTree () {}

  public T root () {
    throw new IllegalArgumentException ("Empty tree");
  }

  public Tree<T> left () {
    throw new IllegalArgumentException ("Empty tree");
  }

  public Tree<T> right  () {
    throw new IllegalArgumentException ("Empty tree");
  }
}


class NodeTree<T> extends Tree<T> {

  private T value;
  private Tree<T> left;
  private Tree<T> right;
```

```
    public NodeTree (T val, Tree<T> l, Tree<T> r) {
      this.value = val;
      this.left = l;
      this.right = r;
    }

    public T root () {
      return this.value;
    }

    public Tree<T> left () {
      return this.left;
    }

    public Tree<T> right () {
      return this.right;
    }
  }
```

A tree transformer now needs to be parameterized not only over the type of values to return from the traversal, but also with the type of trees it should transform.

```
  public interface TreeTransformer<T,U> {
    public U transform (EmptyTree<T> t);
    public U transrorm (NodeTree<T> t, U u1, U u2);
  }
```

All of the above tree transformers can be easily converted to work with this interface by instantiating `T` to `Integer`, because those transformers worked on integer trees. For instance:

```
  public class HeightTransformer implements TreeTransformer<Integer,Integer> {

    public HeightTransformer () { }

    public Integer transform (EmptyTree<Integer> t) {
      return 0;
    }

    public Integer transform (NodeTree<Integer> t, Integer l, Integer r) {
      return 1 + Math.max(l,r);
    }
  }
```

or

```
  public class DoubleTransformer
                   implements TreeTransformer<Integer,Tree<Integer>> {

    public DoubleTransformer () {  }

    public Tree<Integer> transform (EmptyTree<Integer> t) {
      return t;
    }

    public Tree<Integer> transform (NodeTree<Integer> t,
                                    Tree<Integer> l,
                                    Tree<Integer> r) {
      return Tree.node (t.root() * 2, l, r);
    }
  }
```

But now, we can do more. We can, for instance, transform a tree of integers into a tree of strings. For instance, take the following transformer:

```
  public class StringTransformer
                 implements TreeTransformer<Integer,Tree<String>> {

    public StringTransformer () { }

    public Tree<String> transform (EmptyTree<Integer> t) {
      return Tree.empty();
    }

    public Tree<String> transform (NodeTree<Integer> t,
                                   Tree<String> l,
                                   Tree<String> r) {
      return Tree.node("value of this node = " + root().toString(), l, r);
    }
  }
```

Of course, in order to use these transformers, we need to add `reduce` methods to the generic tree code. We could, as we did before, add overloaded `reduce` methods, distinguished by the type of their parameters. But given that are we are biting the genericity bullet, we may as well go the whole way and really take advantage of genericity. If you look at the code for the two overloaded forms of `reduce` we had earlier, we see that aside from the types, the code for the methods is exactly the same in the `EmptyTree` case and in the `NodeTree` case.

That's because the methods are really doing exactly the same thing, they only different in the type of the arguments that are meant to manipulate. Because of this, we can write a single generic method that works "at all types", as follows:

```
public abstract Tree<T> {

  ...

  public abstract <U> U reduce (TreeTransformer<T,U> tr);
}


public EmptyTree<T> extends Tree<T> {

  ...

  public <U> U reduce (TreeTransformer<T,U> tr) {
      return tr.transform (this);
  }
}


public NodeTree<T> extends Tree<T> {

  ...

  public <U> U reduce (TreeTransformer<T,U> tr) {
      U lval = left.reduce(tr);
      U rval = right.reduce(tr);
      return tr.transform(this,lval,rval);
  }
}
```

Despite the apparent complexity, using this is as trivial as using the previous interfaces. Witness:

```
Tree<Integer> e = Tree.empty();
Tree<Integer> t = Tree.node(99, Tree.node(66, e,e),
                                Tree.node(33, e,e));
Tree<String> t2 = t.reduce (new StringTransformer());
```

which yields a new tree `t2` with three nodes, respectively valued `"value of this node = 99"`, `"value of this node = 66"`, and `"value of this node = 33"`.