Design Patterns

A couple of lectures ago, we saw some design pattern to help us enforce statc properties, such as the Singleton pattern, or the Immutability pattern.

Most design patterns, however, play the role of recipes for implementing a certain pattern of behaviors.

Iterator Design Pattern

We already saw one such pattern, the Iterator design pattern, whose purpose is to implement the behavior "provide sequential access to all the entries in an aggregate structure".

The iterators we have studied were functional, via the following interface:

```
public interface FunIterator<T> {
  public boolean hasNext ();
  public T next ();
  public FunIterator<T> advance ();
}
```

The fact that these iterators were functional is reflected in the interface by having an explicit method to make the iterator advance to the next element. In particular, calling the `next` method repeatedly always returns the same answer. Putting it differently, functional iterators are immutable.

Java Iterators

Now, you should all be aware that Java has a built-in notion of iterators, that the aggregator classes in the Java Collections Framework all implement.

These iterators are similar in spirit to the functional iterators, and therefore are an instance of the Iterator design pattern, but they are mutable. Here is the interface Java implements:

```
public interface Iterator<T> {
  public boolean hasNext ();
  public T next ();
}
```

There is no explicit `advance` method. Instead, whenever the `next` method is invoked, it not only returns the next object in the iteration, it also automatically advances the iterator to the next object. Thus, invoking `next` twices on the same iterator may yield two different values.

Part of the advantage of Java iterators is that Java offers syntactic support for them. Consider how you would use a Java iterator, for instance to iterate over all the elements of a `LinkedList<Integer>`?

```
  LinkedList<Integer> lst = ...  // some code to create a list

  Iterator<Integer> it = lst.iterator();
  while (it.hasNext()) {
    Integer nxt = it.next();
    // some code acting on each integer in the list, e.g.:
    System.out.println ("Entry = " + nxt.toString());
  }
```

or, equivalently, using a for-loop:

```
  LinkedList<Integer> lst = ...  // some code to create a list

  for (Iterator<Integer> it = lst.iterator(); it.hasNext(); ) {
    Integer nxt = it.next();
    // some code acting on each integer in the list, e.g.:
    System.out.println ("Entry = " + nxt.toString());
  }
```

(Note that the for-loop does not have any update statement—the update is done implicitly in the call to `next`.)

A class implements `Iterable<T>` if it has a method `iterator` that can return an iterator for the class. Class `LinkedList<T>`, for instance, implements `Iterable<T>`. When you have a class implementing `Iterable<T>`, we can use a special form of a for-loop, called a for-each-loop, as follows:

```
  LinkedList<Integer> lst = ... // some code to create a list

  for (Integer item : lst)
    System.out.println ("Entry = " + item.toString());
```

This is exactly equivalent to the for-loop above.

You can think of the Java statement

```
for (TYPE ITEM_VAR : COLL_EXP)
  STATEMENT
```

as shorthand for the longer

```
for (Iterator<TYPE> it = COLL_EXP.iterator(); it.hasNext(); ) {
  TYPE ITEM_VAR = it.next();
  STATEMENT;
}
```

Adapter Design Pattern

Having two kinds of iterators around can lead to problem. For instance, you might care about using a library that implements a functional iterator, while your code has utility procedures that use Java iterators, or vice versa.

It is of course possible to rewrite code, or reimplement the libraries to use the kind of iterators you code is expecting.

Another possibility is to write a little class that *adapts* the objects returned by the library to interact better with your application. This is a general enough occurrence that we often call this an *Adapted design pattern*, although it is so obvious that the term seems like overkill.

So let's write an adaptor class that wraps around a functional iterator and produces a Java iterator. First, let's define a wrapper class that wraps a Java iterator around a functional iterator:

```
public class IteratorWrapper<T> implements Iterator<T> {

  private FunIterator<T> iter;

  public IteratorWrapper (FunIterator<T> funIterator) {
    iter = funIterator;
  }

  public boolean hasNext () {
    return iter.hasNext();
  }

  public T next () {
    T result = iter.hasNext();
    iter = iter.advance();
    return result;
  }
}
```

Note the implementation of `next`, which needs to update the field holding the functional iterator so that on the next call to `next` another object is extracted from the iteration. You should understand the above code, perhaps using the framework we described in the last lectures about how to model mutability. To use this adapter class, we only need to create an instance of it passing in a functional iterator. Consider the stack examples we have developed in the course, for which we supplied many different functional iterators available through a `getFunIterator` method. If `st` is an integer stack that has been constructed, then the following code gives us a Java iterator on a stack:

```
Iterator<Integer> it = new IteratorWrapper(st.getFunIterator());
```

If additionally we want to be able to use the Java syntax above, we could define an adapter class that wraps around the iterator wrapper and implements the `Iterable` interface, like so:

```
public class IteratorAdapter<T> implements Iterable<T> {

  private FunIterator<T> iter;

  public IteratorAdapter (FunIterator<T> funIterator) {
    this.iter = funIterator;
  }

  public Iterator<T> iterator () {
    return new IteratorWrapper (iter);
  }

}
```

(Interesting question: why did I decide to store the functional iterator in the field `iter`, as opposed to create the Java iterator once and for all at the time when `IteratorAdapter` is invoked and wrapping a new `IteratorWrapper` every time `iterator` is invoked? If you get this, you have nothing more to learn about mutation.)

With the above adapter class, we can now use Java syntax to happily iterate over stacks:

```
Stack st = ... // some code to create an integer stack

for (Integer item : new IteratorAdapter(st.getFunIterator()))
  System.out.println ("Entry = " + item.toString());
```

Exercise: can you write a wrapper class that turns a Java iterator into a functional iterator?

Visitor Design Pattern

Iterators are a great way to traverse an aggregate structure, that is, a structure which is just a collection of other objects.

A related problem is one of traversing a *structured object*. What do I mean by a structured object? It is an object made up of sub-objects, themselves possibly made of up of sub-objects as well, and so on.

The classical example of this in the literature is a car object. Suppose we wanted to implement a class representing a car. This could be useful, for instance, in an application designed to help engineers design new cars. Let's keep thing simple for now, but you can imagine a car as being made up of a number of parts, for instance, four wheels, a chassis, an engine, a steering column, seats. Each wheel is made up of a tire and an axle connector. The seats are made of of stuffing and covering fabric. And so on. As you see, a single car is structured, and each element of the structure can be represented as its own structured object.

The problem: how do you traverse such a structured object, for instance, to perform an operation such as printing data at every element of the structure?

We will call the design we derive a `Visitor` design pattern, because it will give us a way to *visit* every element of a structured object and do something interesting at every node.

And the above sentence gives a hint as to how we will approach things. We will decouple:

(1) the act of traversing the structured object, and

(2) the action to perform at every element.

To traverse the structured object, we will implement, in every class that can be part of the structure of the object, a method `accept` that will be in charge of traversing that part of the structure, possibly invoking `accept` on sub-objects of the structure.

Let's look at how this is done for the `Rect` example of a couple of lectures ago. The example is near trivial, but it illustrates the concept.

```
public class Interval {

  private int lower;
  private int higher;

  ...

  public void accept (...) {
  }
}
```

```
public class Rect {

  private Interval width;
  private Interval height;

  ...

  public void accept (...) {
    this.width.accept(...);
    this.height.accept(...);
  }
}
```

Let's not worry about what we pass to `accept` for now. Just pay attention to the structure. When traversing a `Rect`, we simply in turn traverse both `Interval`s in it.

This takes care of the traversal part. Now we would like to describe what to do at each step of the traversal. To do so, we are going to pass a structure to `accept` that will describe what to do at every element of the structure. That structure will be an instance of the following `ShapeVisitor` interface:

```
public interface ShapeVisitor {

  public void visit (Interval obj);
  public void visit (Rect obj);
}
```

A `ShapeVisitor` gives functions (one per class that we can traverse) that describe what to do at any point of the traversal. We can now update the traversal code to carry such a visitor object around, and invoke it at the appropriate points.

```
public class Interval {

  private int lower;
  private int higher;

  ...

  public void accept (ShapeVisitor visitor) {
    visitor.visit(this);
  }
}
```

```
public class Rect {

  private Interval width;
  private Interval height;

  ...

  public void accept (ShapeVisitor visitor) {
    visitor.visit(this);
    this.width.accept(visitor);
    this.height.accept(visitor);
  }
}
```

Note that at every element we traverse, we invoke the visitor's visit method on the current element. (The types will ensure that Java will resolve the overloading and invoke the right method.)

As an example, consider the following (again, near trivial) visitor for shapes:

```
public class PrintShapeVisitor implements ShapeVisitor {

  public PrintShapeVisitor () { }

  public void visit (Interval obj) {
    System.out.println (" Visiting interval [" + obj.getLower() + " , " +
                           obj.getHigher() + "]");
  }

  public void visit (Rect obj) {
    System.out.println ("Visiting rectangle");
  }
}
```

We can now traverse a `rect` object of type `Rect` by invoking:

```
rect.accept (new PrintShapeVisitor());
```

which produces output like:

```
Visiting rectangle
```

```
 Visiting interval [0,10]
 Visiting interval [0,20]
```

Let's look at a less trivial example. Recall the integer binary trees I asked you to implement on the midterm, with the following signature and algebraic specification.

```
public static Tree empty ();
public static Tree node (int, Tree, Tree);
public boolean isEmpty ();
public int root ();          // extract value at root node
public Tree left ();         // extract left subtree of node
public Tree right ();        // extract right subtree of node


Tree.empty().isEmpty() = true
Tree.node(v,l,r).isEmpty() = false

Tree.node(v,l,r).root() = v
Tree.node(v,l,r).left() = l
Tree.node(v,l,r).right() = r
```

Let's implement a visitor for trees as derived from our recipe. (Note that a tree is a structured object in that implementation, because a `NodeTree` has two fields, one per subtree.) We implement an `accept` method that takes a `TreeVisitor` as input and traverses the tree.

```
public abstract class Tree {

  ...

  public abstract void accept (TreeVisitor visitor);
}


public class EmptyTree extends Tree {

  ...

  public void accept (TreeVisitor visitor) {
    return visitor.visit(this);
  }
}
```

```
public class NodeTree extends Tree {

  private Tree left;
  private Tree right;

  ...

  public void accept (TreeVisitor visitor) {
    this.left.accept(visitor);
    this.right.accept(visitor);
    visitor.visit(this);
  }
}
```

(Note the order in which we traverse the subtrees, and then visit the node itself; this is called a *postorder* traversal of the tree.)

Here is the `TreeVisitor` interface:

```
public interface TreeVisitor {

  public void visit (EmptyTree t);
  public void visit (NodeTree t);
}
```

Let's implement a couple of visitors. First off, the standard print-while-traversing visitor.

```
public class PrintVisitor implements TreeVisitor {

  public PrintVisitor () { }

  public void visit (EmptyTree t) { }

  public void visit (NodeTree t) {
    System.out.println(" Node = " + t.root().toString());
  }
}
```

Let's look at an example.

```
Tree e = Tree.empty ();
Tree t = Tree.node (99, Tree.node (66, e,e),
Tree.node (33, e,e));
```

```
t.accept(new PrintVisitor());
```

Upon execution, this outputs

```
Node = 66
Node = 33
Node = 99
```

Let's implement a more interesting visitor, one that computes the height of a tree. We will have the visitor maintain a current count of the minimum encountered while traversing, and updating it at each node encountered that has a smaller value. (This is ugly and requires mutation, but it is a limitation of the visitor interface we defined that we will address in the next lecture.)

```
public class MinVisitor implements TreeVisitor {

  public int min;
  public boolean found;

  public MinVisitor () {
    this.min = 0;
    this.found = false;
  }

  public int getMin () {
    if (this.found)
      return this.min;
    throw new IllegalArgumentException ("Minimal value not found");
  }

  public void visit (EmptyTree t) {
  }

  public void visit (NodeTree t) {
    if (!this.found) {
      this.found = true;
      this.min = t.root();
    }
    if (t.root() < this.min)
      this.min = t.root();
  }
}
```

The code has some subtleties involving the fact that an empty tree has no minimum value, and therefore we need to keep track of whether we have encountered a minimum value already, or not. (Why can't we initialize the minimum value to be the smallest possible integer? What possible problem can you envision if you do that?)

If we consider the above sample tree `t`, then we can compute:

```
  MinVisitor min = new MinVisitor();
  t.accept(min);
  System.out.println("Height of tree = " + min.getMin().toString());
```

Voilà!