Design Pattern: Adapters

Design patterns, roughly, play the role of recipes for implementing a certain pattern of behaviors.

We already saw one such pattern, the Functional 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 hasElement ();
  public T current ();
  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 `current` method repeatedly always returns the same answer. Putting it differently, functional iterators are immutable.

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 a form of Iterator design pattern, but they are mutable. Here is the interface Java implements:

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

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.

Diversion: why do we care?

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

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 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 while loop above. In fact, you can think of the Java statement

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

as shorthand for the longer

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

Exercise 1: *Take the Stack ADT we've been looking at the whole semester, or the Grid ADT from homework 2, and add a method:*

```
public Iterator<Integer> iterator ();
```

(for Stack), or

```
    public Iterator<Line> iterator ();
```

(for Grid) that creates a Java iterator for the instance at hand.

Having two kinds of iterators around is a bit of a mess. 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 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 IteratorAdapter<T> implements Iterator<T> {

    private FunIterator<T> iter;

    private IteratorAdapter (FunIterator<T> funIterator) {
      iter = funIterator;
    }

    public static <U> IteratorAdapter<U> create (FunIterator<U> it) {
      return new IteratorAdapter<U>(it);
    }

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

    public T next () {
      T result = iter.hasNext();
      iter = iter.advance();      // update the underlying iterator!
      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

3

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 = IteratorAdapter.create(st.getFunIterator());
```

Exercise 2: *Can you write an adapter class that turns a Java iterator into a functional iterator? Here is the skeleton:*

```java
public class FunIteratorAdapter<T> implements FunIterator<T> {

  ...

  private FunIteratorAdapter (Iterator<T> iterator) {
    ...
  }

  public static <U> FunIteratorAdapter<U> create (Iterator<U> it) {
    return new FunIteratorAdapter<U>(it);
  }

  public boolean hasElement () {
    ...
  }

  public T current () {
    ...
  }

  public FunIterator<T> advance () {
    ...
  }
}
```

Exercise 3 (advanced, not given in class): *Can you figure out a way (there are many) to get the iterators that we get out of the adapter to play nicely with the convenient Java syntax for iteration that I gave earlier?*

Design Pattern: Visitors

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 `traverse` that will be in charge of traversing that part of the structure, possibly invoking `traverse` on sub-objects of the structure.

Let's look at how this is done for the Line and Point ADTs from early in the course. The example is near trivial, but it illustrates the concept.

```
public class Point {

  private int xpos;
  private int ypos;

  ...

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

```
public class Line {

  private Point start;
  private Point end;

  ...

  public void traverse (...) {
    this.start.traverse(...);
    this.end.traverse(...);
  }
}
```

Let's not worry about what we pass to `traverse` for now. Just pay attention to the structure. When traversing a `Line`, we simply in turn traverse both `Point`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 `traverse` that will describe what to do at every element of the structure. That structure will be an instance of the following `GeometryVisitor` interface:

```
public interface GeometryVisitor {

  public void visitPoint (Point obj);
  public void visitLine (Line obj);
}
```

(We could potentially simplify this in Java by *overloading* the method name `visit` for both functions.)

A `GeometryVisitor` gives methods (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 Point {

  private int xpos;
  private int ypos;

  ...

  public void traverse (GeometryVisitor visitor) {
```

```
      visitor.visitPoint(this);
    }
  }


public class Line {

  private Point start;
  private Point end;

  ...

  public void traverse (GeometryVisitor visitor) {
    visitor.visitLine(this);
    this.start.traverse(visitor);
    this.end.traverse(visitor);
  }
}
```

Note that at every element we traverse, we invoke the visitor's visit method on the current element.

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

```
public class PrintVisitor implements GeometryVisitor {

  private PrintVisitor () { }

  public static PrintVisitor create () {
    return new PrintVisitor();
  }

  public void visitPoint (Point obj) {
    System.out.println(" Visiting point [" + obj.xPos() + " , " +
                       obj.yPos() + "]");
  }

  public void visitLine (Line obj) {
    System.out.println("Visiting line");
  }
}
```

We can now traverse a `line` object of type `Line` by invoking:

```
    line.traverse(PrintVisitor.create());
```

which produces output like (the details depend on the actual line):

```
  Visiting line
   Visiting point [0,10]
   Visiting point [0,20]
```

Not covered in class

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 a `traverse` method that takes a `TreeVisitor` as input and traverses the tree.

```
  public abstract class Tree {

    ...

    public abstract void traverse (TreeVisitor visitor);
  }


  public class EmptyTree extends Tree {
```

```
    ...

    public void traverse (TreeVisitor visitor) {
      return visitor.visitEmpty(this);
    }
  }


  public class NodeTree extends Tree {

    private Tree left;
    private Tree right;

    ...

    public void traverse (TreeVisitor visitor) {
      this.left.traverse(visitor);
      this.right.traverse(visitor);
      visitor.visitNode(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 visitEmpty (EmptyTree t);
    public void visitNode (NodeTree t);
  }
```

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

```
  public class PrintVisitor implements TreeVisitor {

    private PrintVisitor () { }

    public static PrintVisitor create () {
      return new PrintVisitor();
    }
```

```
    public void visitEmpty (EmptyTree t) { }

    public void visitNode (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.traverse(PrintVisitor.create());
```

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.)

```
public class MinVisitor implements TreeVisitor {

  public int min;

  private MinVisitor (int init) {
    this.min = init;
  }

  public MinVisitor create (int init) {
    return new MinVisitor(init);
  }

  public int getMin () {
    return this.min;
  }
```

```
  public void visitEmpty (EmptyTree t) {
  }

  public void visitNode (NodeTree t) {
    if (t.root() < this.min)
      this.min = t.root();
  }
}
```

We initialize the visitor with a default minimum value. (Why?)

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

```
MinVisitor min = MinVisitor.create(0);
t.traverse(min);
System.out.println("Height of tree = " + min.getMin().toString());
```

Voilà!