18 Design Pattern: Adapters

Recall that design patterns, roughly, play the role of recipes for implementing a certain pattern of behaviors.

> **(Small aside)** The book that first introduced software design patterns to the world, and that remains the best reference on the topic, is "Design Patterns: Elements of Reusable Object-Oriented Software" (Addison-Wesley Professional Computing Series), by Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, also known as "The Gang of Four".
>
> It was the first book to "formalize" the idea of design patterns for software, and, probably even more helpful, gave a taxonomy of patterns for the working programmer. All patterns in the book come with sample code, but the code is in C++. However, once you know the design pattern you need, you can look up online and find a Java or C# version pretty easily.

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".

Recall that our functional iterators use 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. (The term *functional* often carries the meaning of *immutable* in programming.)

18.1 Java Iterators

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

Java iterators are similar in spirit to the functional iterators, and therefore are a form of Iterator design pattern. The big difference is that they are *mutable*, which is why sometimes we'll call them *mutable iterators*. Here is the interface Java implements (in `java.util.Iterator`):

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

Method `hasNext()` is similar to `hasElement()` in functional iterators, it returns whether there is an element left to return from the iteration. Method `next()` returns the next element in the iteration, like `current`, except is also automatically advances the iterator to the next object. The final method `remove()`, can be used for removing elements during iteration, but we won't be bothering with it. It is optional anyways according to the Java documentation, and an implementation is free to throw the `UnsupportedOperationException` when `remove()` is called.

Calling `next` does not produce a new instance of the iterator to point to the next element, unlike functional iterators: the current instance of the iterator is mutated so that it now points to the next element. This means, among other things, that invoking `next` twices on the same iterator will generally yield two different values.

Why do we care about Java iterators at all, aside from the fact that we need to know about them to iterate over Java Collections classes? Putting it differently, if you already have functional iterators for a particular ADT you've designed, why should you care about Java iterators?

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 `List<T>` ADT we've been looking at the whole semester, and add a method:*

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

that creates a Java iterator for the instance at hand, and also implement that iterator

18.2 An Adapter for Iterators

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 *Adapter design pattern*, although it is so obvious that the term seems like overkill.

Let's start with a `List<A>` ADT that also implements functional iterators, from Lecture 10:

```
public abstract class List<A> implements Iterable<A> {

   public static <A> List<A> empty () {
      return new EmptyList<A>();
   }
   public static <A> List<A> cons (A v, List<A> l) {
      return new ConsList<A>(v,l);
   }

   public abstract boolean isEmpty ();
   public abstract A first ();
   public abstract List<A> rest ();
   public abstract String toString ();

   public abstract FuncIterator<A> getFuncIterator ();
}


class EmptyList<A> extends List<A> {
   public EmptyList () {}

   public boolean isEmpty () { return true; }

   public A first () { throw new RuntimeException("EmptyList.first()"); }

   public List<A> rest () { throw new RuntimeException("EmptyList.rest()"); }

   public String toString () { return ""; }

   public FuncIterator<A> getFuncIterator () {
      return new EmptyFuncIterator<A>();
   }
}


class EmptyFuncIterator<A> implements FuncIterator<A> {
   public EmptyFuncIterator () {}

   public boolean hasElement () { return false; }

   public A current () {
        throw new java.util.NoSuchElementException("EmptyFuncIterator.current()");
    }

    public FuncIterator<A> advance () {
        throw new java.util.NoSuchElementException("EmptyFuncIterator.advance()");
    }
}


class ConsList<A> extends List<A> {
    private A first;
    private List<A> rest;

    public ConsList (A f, List<A> r) {
        this.first = f;
        this.rest = r;
    }

    public boolean isEmpty () { return false; }

    public A first () { return this.first; }

    public List<A> rest () { return this.rest; }

    public String toString () { return this.first() + " " + this.rest(); }

    public FuncIterator<A> getFuncIterator () {
        return new ConsFuncIterator<A>(this.first(),
                                       this.rest().getFuncIterator());
    }
}


class ConsFuncIterator<A> implements FuncIterator<A> {
    private A current;
    private FuncIterator<A> rest;

    public ConsFuncIterator (A c, FuncIterator<A> r) {
        this.current = c;
        this.rest = r;
    }
```

```
  public boolean hasElement () { return true; }

  public A current () { return this.current; }

  public FuncIterator<A> advance () { return this.rest; }
}
```

We could add a `iterator()` method to `List<A>` that returns an `Iterator<A>` instance, as per Exercise 1 above, but this would require us to rewrite a lot of iterator-related code, and we already have functional iterators implemented for lists. So let's write an adapter class that wraps around a functional iterator and gives back a Java iterator that iterates over the same values as the functional iterator.

```
import java.util.Iterator;

public class IteratorAdapter<A> implements Iterator<A> {
  private FuncIterator<A> fIter;

  private IteratorAdapter (FuncIterator<A> f) {
    this.fIter = f;
  }

  public static <A> IteratorAdapter<A> create (FuncIterator<A> f) {
    return new IteratorAdapter<A>(f);
  }

  public boolean hasNext () {
    return fIter.hasElement();
  }

  public A next () {
    if (!(fIter.hasElement()))
      throw new java.util.NoSuchElementException("IteratorAdapter.next()");
    A current = fIter.current();
    fIter = fIter.advance();
    return current;
  }

  public void remove () {
    throw new UnsupportedOperationException("IteratorAdapter.remove()");
  }
}
```

First, note that the above adapter is not specific to functional iterators over lists — it works for *any* functional iterator, and turns any such functional iterator into a Java iterator. Second, 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 I described last lecture for modeling mutability. To use this adapter class, we only need to create an instance of it, passing in a functional iterator. In particular, we can revise our `List<A>` abstract class to both provide an `iterator` method, and to tell Java that `List<A>` provides such a method:

```
import java.util.Iterator;
import java.lang.Iterable;

public abstract class List<A> implements Iterable<A> {
   public static <A> List<A> empty () {
      return new EmptyList<A>();
   }
   public static <A> List<A> cons (A v, List<A> l) {
      return new ConsList<A>(v,l);
   }

   public abstract boolean isEmpty ();
   public abstract A first ();
   public abstract List<A> rest ();
   public abstract String toString ();

   public abstract FuncIterator<A> getFuncIterator ();

   public Iterator<A> iterator () {
      return IteratorAdapter.create(this.getFuncIterator());
   }
}
```

We can try out this code as follows. First, define a function to print the elements

```
  public static <A> void elements (FuncIterator<A> fIter) {
    FuncIterator<A> it = fIter;
    while (it.hasElement()) {
      System.out.println(" Element = " + it.current());
      it = it.advance();
    }
  }
```

Next, some sample code, construct a list and iterating over it using both its functional iterator, and its Java iterator:

```
  List<String> el = List.empty();
  List<String> l = List.cons("goodbye",List.cons("cruel",List.cons("world",el)));
  System.out.println("l = " + l);
  System.out.println("--------------------------------------------------");
  System.out.println("Loop using functional iterator:");
  FuncIterator<String> it = l.getFuncIterator();
  elements(it);
  System.out.println("Testing immutability:");
  System.out.println(" it.current() = " + it.current());
  System.out.println(" it.current() = " + it.current());
  System.out.println("--------------------------------------------------");
  System.out.println("Loop using Java iterator:");
  for (String s : l)
    System.out.println(" Element = " + s);
  System.out.println("Testing mutability:");
  Iterator<String> it3 = l.iterator();
  System.out.println(" it3.next() = " + it3.next());
  System.out.println(" it3.next() = " + it3.next());
```

Which yields:

```
l = goodbye cruel world
--------------------------------------------------
Loop using functional iterator:
 Element = goodbye
 Element = cruel
 Element = world
Testing immutability:
 it.current() = goodbye
 it.current() = goodbye
--------------------------------------------------
Loop using Java iterator:
 Element = goodbye
 Element = cruel
 Element = world
Testing mutability:
 it3.next() = goodbye
 it3.next() = cruel
```

18.3 An Adapter for Functional Iterators

The above shows that we can adapt a functional iterator into a Java iterator. How about the other way around? More precisely, suppose we have an application geared towards using functional iterators, and we want to use one of the Java Collections classes, which all implement Java iterators.

It turns out to be a bit more complicated to adapt Java iterators into functional iterators, and not very safe, because as we saw last time, mutability is contagious. Here is one stab at such an adapter:

```
import java.util.*;

public class FuncIteratorAdapter<A> implements FuncIterator<A> {
   private Option<A> current;
   private final Iterator<A> iter;

   public FuncIteratorAdapter (Iterator<A> i) {
      if (i.hasNext())
         current = Option.some(i.next());
      else
         current = Option.none();
      this.iter = i;
   }

   public static <A> FuncIteratorAdapter<A> create (Iterator<A> i) {
      return new FuncIteratorAdapter<A>(i);
   }

   public boolean hasElement () {
      return !(this.current.isNone());
   }

   public A current () {
      if (this.current.isNone())
         throw new NoSuchElementException("FuncIteratorAdapter.current()");
      return this.current.valOf();
   }

   public FuncIterator<A> advance () {
      return new FuncIteratorAdapter<A>(this.iter);
   }
}
```

This adapter uses the Option ADT, which we saw in Lecture 12 as well as on the last homework. The Option ADT lets us avoid the use of null, which is always error-prone.

The idea behind the above adapter is that we have to make sure that we never call next() more than once for every element of the underlying Java iterator, because whenever we call next(), the iterator mutates. We therefore call next() once when we construct the functional iterator, storing the value (if there is one — if there is none, we also record that fact, using Option.none()) away so that when we ask for current() we do not need to query the underlying iterator, we just return the value we stored away. When we advance the iterator, we simply construct a new adapter for the underlying iterator, which by that point is already pointing to the next element to be returned anyways.

Again, stare at the above code, and convince yourself that it works, and try to understand that it works. When you do, then you have nailed how mutation works.

Let's test the above adapter by constructing a functional iterator from the Java iterators of our List<A> implementation (which themselves were constructed from functional iterators, so that's many layers of indirection here), continuing on with the example from last section:

```
  System.out.println("Loop using (converted) functional iterator:");
  Iterator<String> it4 = l.iterator();
  FuncIterator<String> it5 = FuncIteratorAdapter.create(it4);
  elements(it5);
  System.out.println("Testing immutability:");
  System.out.println(" it5.current() = " + it5.current());
  System.out.println(" it5.current() = " + it5.current());
```

which yields:

```
Loop using (converted) functional iterator:
 Element = goodbye
 Element = cruel
 Element = world
Testing immutability:
 it5.current() = goodbye
 it5.current() = goodbye
```

I claimed above that FuncIteratorAdapter is not safe. What I mean is that there are ways to disrupt the result of iterating using a functional iterator obtained by FuncIteratorAdapter. The underlying Java iterator is shared with the resulting functional iterator, meaning that we can mutate the underlying Java iterator, and the result of the mutation will be visible in the functional iterator. Study the following example, again continuing the sample code above:

```
  System.out.println("Messing with iteration:");
  Iterator<String> it6 = l.iterator();
  FuncIterator<String> it7 = FuncIteratorAdapter.create(it6);
  System.out.println(" it7.current() = " + it7.current());
  System.out.println("Advancing underlying iterator -- " + it6.next());
  System.out.println(" it7.advance().current() = " + it7.advance().current());
```

This yields:

```
Messing with iteration:
 it7.current() = goodbye
Advancing underlying iterator -- cruel
 it7.advance().current() = world
```

By accessing the Java iterator `it6`, we made the `it7` functional iterator skip an element. That's bad.

Exercise 2: *Write a different `FuncIteratorAdapter<A>` class that does not have this problem.*

18.4 Iterators for Trees

Adapting is useful to bridge gaps between code that provides a particular interface, and client code that expects a slightly different interface. Functional iterators and Java iterators being just one example of such bridging that is sometimes necessary.

Another use for adapters is that it sometimes helps simplify coding. Consider binary trees, of the kind we saw in an exercise in Lecture 12, with signature

```
  CREATORS:   <A> Tree<A> empty ()
              <A> Tree<A> node (A, Tree<A>, Tree<A>)

  ACCESSORS:  boolean isEmpty ()
              A root ()
              Tree<A> left ()
              Tree<A> right ()
              String toString ()
```

and the obvious specification. Defining Java iterators for trees is difficult — try it. Pick an order in which to traverse the tree (there are a few), and define an `iterator()` method for the implementation of trees you obtain by applying our Specification design pattern to the above specification.

There is another way to get Java iterators for trees, though. It is still difficult, but much easier, to define a functional iterator for trees. Once we have functional iterators, it suffices to use the `IteratorAdapter` class above to get Java iterators. Here is one implementation of functional iterators for trees, that implements a preorder iteration of the tree.[1]

```
import java.util.*;
import java.lang.Iterable;

public abstract class Tree<A> implements Iterable<A> {
  public static <A> Tree<A> empty () {
    return new EmptyTree<A>();
  }
  public static <A> Tree<A> node (A v, Tree<A> l, Tree<A> r) {
    return new NodeTree<A>(v,l,r);
  }

  public abstract boolean isEmpty ();
  public abstract A root ();
  public abstract Tree<A> left ();
  public abstract Tree<A> right ();
  public abstract String toString ();

  public abstract FuncIterator<A> getFuncIterator ();

  public Iterator<A> iterator () {
    return IteratorAdapter.create(this.getFuncIterator());
  }
}


class EmptyTree<A> extends Tree<A> {
  public EmptyTree () {}

  public boolean isEmpty () { return true; }

  public A root () { throw new RuntimeException("EmptyTree.root()"); }

  public Tree<A> left () { throw new RuntimeException("EmptyTree.left()"); }

  public Tree<A> right () { throw new RuntimeException("EmptyTree.right()"); }
```

[1]A preorder iteration says that we return the value of a node before we iterate over its left or right subtrees. That's in contrast with a postorder iteration, where we first iterate over the left and right subtrees, then we return the value of the node, or an inorder iteration, where we first iterate over the left subtree, return the value of the node, then iterate over the right subtree.

```
  public String toString () { return ""; }

  public FuncIterator<A> getFuncIterator () {
    return new EmptyTFuncIterator<A>();
  }
}


class EmptyTFuncIterator<A> implements FuncIterator<A> {
  public EmptyTFuncIterator () {}

  public boolean hasElement () { return false; }

  public A current () {
    throw new NoSuchElementException("EmptyTFuncIterator.current()");
  }

  public FuncIterator<A> advance () {
    throw new NoSuchElementException("EmptyTFuncIterator.advance()");
  }
}


class NodeTree<A> extends Tree<A> {
  private A root;
  private Tree<A> left;
  private Tree<A> right;

  public NodeTree (A v, Tree<A> l, Tree<A> r) {
    this.root = v;
    this.left = l;
    this.right = r;
  }

  public boolean isEmpty () { return false; }

  public A root () { return this.root; }

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

  public Tree<A> right () { return this.right; }
  public String toString () {
    return "[" + this.left() + "] " + this.root() + " [" + this.right() + "]";
  }

  public FuncIterator<A> getFuncIterator() {
    List<FuncIterator<A>> e = List.empty();
    List<FuncIterator<A>> remainder =
      List.cons(this.right().getFuncIterator(),e);
    return new NodeFuncIterator<A>(this.root(),
                                   this.left().getFuncIterator(),
                                   remainder);
  }
}


class NodeFuncIterator<A> implements FuncIterator<A> {
  private A currentElement;
  private FuncIterator<A> currentIterator;
  private List<FuncIterator<A>> remainder;

  public NodeFuncIterator (A v, FuncIterator<A> n, List<FuncIterator<A>> r) {
    this.currentElement = v;
    this.currentIterator = n;
    this.remainder = r;
  }

  public boolean hasElement () { return true; }

  public A current () { return this.currentElement; }

  public FuncIterator<A> advance () {
    // element left in current iterator
    if (this.currentIterator.hasElement())
      return new NodeFuncIterator<A>(this.currentIterator.current(),
                                     this.currentIterator.advance(),
                                     this.remainder);
    // no element in current iterator -- any iterators remaining?
    if (this.remainder.isEmpty())
      return new EmptyTFuncIterator<A>();
    // find next nonempty iterator
    FuncIterator<A> nextIter = this.remainder.first();
    List<FuncIterator<A>> nextRemainder = this.remainder.rest();
    while (!(nextIter.hasElement())) {
      if (nextRemainder.isEmpty())
        return new EmptyTFuncIterator<A>();
      nextIter = nextRemainder.first();
      nextRemainder = nextRemainder.rest();
    }
    return new NodeFuncIterator<A>(nextIter.current(),
                                   nextIter.advance(),
                                   nextRemainder);
  }
}
```

The difficulty in this code is the implementation of `NodeFuncIterator`, where we iterate over a node. The idea is that we return the value at the node, and then when we advance, we need to return a functional iterator that can yield the rest of the elements in the tree, namely the elements in the left subtree, and the elements in the right subtree. We know how to return an iterator for the elements of the left subtree — we just get the iterator that corresponds to the left subtree by calling `getFuncIterator()` on the left subtree. But we need to remember somewhere that when the iterator for the left subtree is exhausted, we have to look at the elements in the right subtree. And we need to do so recursively as we progress down the tree. So we record in `NodeFuncIterator` the current element in the tree we're at, an "immediate" iterator that can immediately give us new elements, and the remainder of the iterators where we accumulate the list of all iterators that still have elements to yield, essentially corresponding to all the right subtrees we have encountered but not visited yet. (That's a brainful. And I still claim that this is easier than defining a mutable iterator directly.)

Note that `iterator()` in `Tree<A>` just uses our functional iterators and the `IteratorAdapter` class from before.

Here are the iterators in action:

```
Tree<Integer> et = Tree.empty();
Tree<Integer> t66 = Tree.node(66,et,et);
Tree<Integer> t87 = Tree.node(87,et,et);
Tree<Integer> t = Tree.node(99,t66,t87);
System.out.println("t = " + t);
System.out.println("--------------------------------------------------");
System.out.println("Loop using functional iterator:");
FuncIterator<Integer> itt = t.getFuncIterator();
elements(itt);
System.out.println("--------------------------------------------------");
System.out.println("Loop using Java iterator:");
```

```
  for (Integer i : t)
    System.out.println(" Element = " + i);
```

(where we use the `elements` function defined in §18.2), which yields:

```
t = [[] 66 []] 99 [[] 87 []]
--------------------------------------------------
Loop using functional iterator:
 Element = 99
 Element = 66
 Element = 87
--------------------------------------------------
Loop using Java iterator:
 Element = 99
 Element = 66
 Element = 87
```