Design Pattern: Functional Iterators

Subclassing enables the use of standardized interfacs for implementing specific functionality. Standardized interfaces enable code reure (on the client side) because it becomes possible to write functions against those standardized interfaces that will work for every subclass of those interfaces.

Let's look at an example. First, some terminology. An *aggregate* is a data type that contains objects, such as a list data type, or a tree data type, or a hash table data type, or a stack data type. Arrays are typical aggregates. Arrays have the added advantage that there is nice built-in syntax for them, including easy ways of iterating over all the elements of an array using a `for` loop.

This idea of iteration can be generalized to all aggregates. An *iterator* is an object whose sole purpose is to make it easy to iterate over all the elements of an aggregate.

What's an iterator? A (functional) iterator (over integers) is an class that subclasses the following class — which we will represent as an interface in Java:

```
public interface FuncIterator {
  public boolean hasElement ();
  public int current ();
  public FuncIterator advance ();
}
```

Functional is often used as a synonym for immutable. A functional iterator provides a method `hasElement()` for asking whether we are done iterating over the elements of the underlying aggregate, a method `current()` for getting the current element of the aggregate we have not looked at yet, and a method `advance()` that advances the iterator past the current element so that we can look at the element after than. The `advance` method returns a new iterator that can be queried for that next element.[1] The point here is that different aggregates will require quite different iterators—there is no notion of a default implementation of an

[1]Java comes with an iterator interface in its basic API that is mutable; it does not have an `advance()` method, and query for the current element in the interface mutates the iterator under the hood so that querying it for the next element returns yet a new element. This mutability, as usual, makes it somewhat more difficult to reason about iteration (but also sometimes more efficient — the usual tradeoff). We will return to mutable iterators later.

iterator that works for all aggregates. It therefore makes sense that we use a Java interface (or equivalently, we could use an abstract class).

Lists are aggregate, so let's define an iterator for lists. (Recall from last lecture.) The first thing we need to do is add a method to the `List` class that gives us a functional iterator to iterate over any instance of `List`. That method creates a new instance of a list iterator, which is a class that we need to define.

To illustrate the use of iterators, if we have an aggregate with a `getFuncIterator` method that gives us back a `FuncIterator`, we can pass such an iterator to the following function, that simply computes the sum of the elements in the aggregate:

```
 public static int computeIterSum (FuncIterator arg) {
   FuncIterator i = arg;
   int total=0;
   while (i.hasElement()) {
     total=total+i.current();
     i=i.advance();
   }
   return total;
 }
```

This will work for lists with a suitably defined iterator, for trees with again a suitably defined iterator, and so on.

There are many ways of implementing iterators. Some are easier than others, depending on the order in which the iterator is meant to deliver the elements of the aggregate. For lists, the easiest order is the natural order of the list. The following code, a variant on the code we produced last time from the design pattern for ADT implementation, implements a straightforward functional iterator for lists.

```
/* ABSTRACT CLASS FOR LISTS */
public abstract class List {

  // no java constructor for this class, it is abstract

  public static List empty () {
    return new EmptyList();
  }

  public static List cons (int i, List l) {
    return new ConsList(i,l);
  }

  public abstract boolean isEmpty ();

  public abstract int first ();

  public abstract List rest ();

  public FuncIterator getFuncIterator () {
    return new ListFuncIterator(this);
  }
}


/* CONCRETE CLASS FOR EMPTY CREATOR */
class EmptyList extends List {

 public EmptyList () {}

  public boolean isEmpty () {
    return true;
  }

  public int first () {
    throw new Error ("first() on an empty list");
  }

  public List rest () {
    throw new Error ("rest() on an empty list");
  }
}


/* CONCRETE CLASS FOR CONS CREATOR */
class ConsList extends List {

  private int firstElement;
  private List restElements;

  public ConsList (int f, List r) {
    firstElement = f;
    restElements = r;
  }
  public boolean isEmpty () {
    return false;
  }

  public int first () {
    return firstElement;
  }

  public List rest () {
    return restElements;
  }
}


/* ITERATOR OVER LISTS IN THE NATURAL ORDER */
class ListFuncIterator implements FuncIterator {

  private List listToIterateOver;

  public ListFuncIterator (List l) {
    listToIterateOver = l;
  }

  public boolean hasElement () {
    return !(listToIterateOver.isEmpty());
  }

  public int current () {
    if (this.hasElement())
      return listToIterateOver.first();
    throw new java.util.NoSuchElementException
                         ("list is empty during iteration");
}

  public FuncIterator advance () {
    if (this.hasElement())
      return new ListFuncIterator(listToIterateOver.rest());
    throw new java.util.NoSuchElementException
                         ("list is empty during iteration");
  }
}
```

(As before, we can nest the `ListFuncIterator` inside the `List` class, just like `EmptyList` and `ConsList`.) Study the above code, and see why it works. In particular, note that the reason why this works at all is that both the iterator *and* the list class are immutable. Because a list, once created, never changes, we can freely pass it around to the iterator, who is free to iterate over it without worrying about some other part of the code changing that list. In effect, this is equivalent to everyone getting their own copy of a list when it is passed to them. The exception thrown when trying to get at the current element when there is no such element, or advancing the iterator passed the end of the list is the one that the Java API requires its iterators to throw, so I have done so here for consistency.

Exercise: *Modify the above code to iterate over a list from the last element to the first.*

Exercise: *Modify the above code to iterate over a list from the first element to the last, but listing all the odd-positioned elements first, then all the even-positioned elements.*

We can do better, though. The above code contains a lot of if-then-else statements that branch based on the representation (whether the list is empty or not). As we saw using the recipe, we can eliminate these kind of if-then-else statements by appropriate use of subclassing. We can do the same here, by have `getFuncIterator` be abstract in the abstract `List` class, and implementing different `getFuncIterator` in each of the concrete subclasses. We will have different classes implementing iterators corresponding to each of those `getFuncIterator` methods.

```
/* ABSTRACT CLASS FOR LISTS */
public abstract class List {

  // no java constructor for this class, it is abstract

  public static List empty () {
    return new EmptyList();
  }

  public static List cons (int i, List l) {
    return new ConsList(i,l);
  }

  public abstract boolean isEmpty ();

  public abstract int first ();

  public abstract List rest ();

  public abstract FuncIterator getFuncIterator ();
}
```

```
/* CONCRETE CLASS FOR EMPTY CREATOR */
class EmptyList extends List {

 public EmptyList () {}

  public boolean isEmpty () {
    return true;
  }

  public int first () {
    throw new Error ("first() on an empty list");
  }

  public List rest () {
    throw new Error ("rest() on an empty list");
  }

  public FuncIterator getFuncIterator () {
    return new EmptyFuncIterator();
  }
}


/* ITERATOR FOR EMPTY LISTS */
class EmptyFuncIterator implements FuncIterator {

  public EmptyFuncIterator () {}

  public boolean hasElement () {
    return false;
  }

  public int current () {
   throw new java.util.NoSuchElementException
                        ("list is empty during iteration");
  }

  public FuncIterator advance () {
   throw new java.util.NoSuchElementException
                        ("list is empty during iteration");
  }
}


/* CONCRETE CLASS FOR CONS CREATOR */
class ConsList extends List {

  private int firstElement;
  private List restElements;

  public ConsList (int f, List r) {
    firstElement = f;
    restElements = r;
  }

  public boolean isEmpty () {
    return false;
  }

  public int first () {
    return firstElement;
  }

  public List rest () {
    return restElements;
  }

  public FuncIterator getFuncIterator () {
    return new ConsFuncIterator(firstElement,
                                restElements.getFuncIterator());
  }
}


/* ITERATOR FOR NON-EMPTY LISTS */
class ConsFuncIterator implements FuncIterator {

  private int currentElement;
  private FuncIterator restIterator;

  public ConsFuncIterator (int c, FuncIterator r) {
    currentElement = c;
    restIterator = r;
  }

  public boolean hasElement () {
    return true;
  }

  public int current () {
    return currentElement;
  }

  public FuncIterator advance () {
    return restIterator;
  }
}
```

It is also possible to nest the `EmptyFuncIterator` class inside the `EmptyList` class, and similarly nesting the `ConsFuncIterator` class inside the `ConsList` class. Or we can just nest everything inside the `List` class, as before.

As a side note, this implementation essentially corresponds to the following algebraic specification for functional iterators over lists:

```
empty().getFuncIterator().hasElement() = false
cons(i,L).getFuncIterator().hasElement() = true

cons(i,L).getFuncIterator().current() = i
cons(i,L).getFuncIterator().advance() = L.getFuncIterator()
```