14 Design Patterns: Map and Reduce

A couple of lectures back, we saw a very important design pattern for ADTs, functional iterators, to get all elements of a structure in some kind of order.

Today, we look at two other design patterns that can be used to capture a large class of behaviors for a large class of ADTs.

Again, we will use lists as our starting point, because as I keep repeating you have some independent intuition for how lists work. But all that I'm saying here applies to other ADTs, as we will soon see.

First, let's remind ourselves of our (polymorphic) lists implementation, where I've dropped the functional iterators, and added an operation `append` and `toString` for reasons that will soon become clear. Their specifications are straightforward:

```
ACCESSORS:     ...
               List<A> append (List<A>)
               String toString ()

SPECIFICATION:  ...
                empty().append(M) = M
                cons(a,L).append(M) = cons(a,L.append(M))
                empty().toString() = ""
                cons(a,L).toString() = a.toString()+"  "+L.toString()
```

which yields the following implementation of lists:

```
public abstract class List<A> {

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

 public abstract boolean isEmpty ();
```

```
  public abstract A first ();
  public abstract List<A> rest ();
  public abstract List<A> append (List<A> l);
  public abstract String toString ();
}


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

  public boolean isEmpty () {
    return true;
  }
  public A first () {
    throw new Error ("EmptyList.first()");
  }
  public List<A> rest () {
    throw new Error ("EmptyList.rest()");
  }
  public List<A> append (List<A> l) {
      return l;
  }
  public String toString () {
      return "";
  }
}


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

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

  public boolean isEmpty () {
    return false;
  }
  public A first () {
    return first;
```

```
  }
  public List<A> rest () {
    return rest;
  }
  public List<A> append (List<A> l) {
     return List.cons(first,rest.append(l));
  }
  public String toString () {
     return first.toString() + "   " + rest.toString();
  }
}
```

14.1 The Map Design Pattern

The first thing we will do is figure out a way to modify a list (or any data structure) uniformly, that is, changing the information stored in the list, without changing the structure of the list. The idea is to *map* an operation over a list, by changing each element of the list. You saw such a map operation in previous courses.

The idea is to have an operation `List<A> map (??)`, for some appropriate `??`, with the following specification:

```
empty().map(??) = empty()
cons(a,L).map(??) = cons( [?? on a], L.map(??))
```

The question is, what do we give to `map` so we can apply it to an element of the list? What do we take as `??` — in Scheme, this would be a function, because functions are first-class. In Java, the only thing first class are objects, so we are going to use an object to represent a function. Roughly speaking, a function can be represented by an object that has one method `apply` that applies the function to an argument. Here is an interface that captures this:

```
public interface MappingFunction<A> {
   public A apply (A arg);
}
```

With that, we can give an actual signature and specification for the `map` operation.

```
ACCESSORS:    ...
              List<A> map (MappingFunction<A> mf);

SPECIFICATION:  ...
```

```
empty().map(mf) = empty()
cons(a,L).map(mf) = cons(mf.apply(a), L.map(mf))
```

Adding this to the list specification and applying the design pattern we have for deriving implementations from ADT specifications, we get the following methods added to the list implementation we gave above:

```
public abstract class List<A> {
  ...
  public abstract List<A> map (MappingFunction<A> mf);
}


class EmptyList<A> extends List<A> {
  ...
  public List<A> map (MappingFunction<A> mf) {
     return List.empty();   // equivalently, return this;
  }
}


class ConsList<A> extends List<A> {
  ...
  public List<A> map (MappingFunction<A> mf) {
     List<A> r = rest.map(mf);
     return List.cons(mf.apply(first),r);
  }
}
```

Things may be a somewhat clearer by looking at examples of of `MappingFunction`s we can use with `map`. First, we construct a sample list to play with:[1]

```
List<Integer> lst =
  List.cons(1,List.cons(2,List.cons(3,List.cons(4,List.<Integer>empty()))));
System.out.println("Initial list = " + lst);
```

with output

[1]Note the interesting call to the `empty()` creator: `List.<Integer>empty()`. This is how you call a polymorphic function when you want to specify the type argument at which you want to instantiate the function. (This is useful when Java cannot figure out the right type to use — here, because `empty()` does not take an argument, Java cannot figure out what kind of empty list you want, so we say explicitly that we want the type parameter of `empty()` to be instantiated at type `Integer`.)

```
Initial list = 1  2  3  4
```

Now, first consider the problem of doubling every element of an integer list. We can create a class whose instances are functions that double an integer.

```
public class Double implements MappingFunction<Integer> {
  private Double () {}

  public static Double function () { return new Double(); }

  public Integer apply (Integer a) { return 2 * a; }
}
```

(Note that I usually use a creator called `function` as a mnemonic reminder that the class is used to represent functions.) Once we have this, we can use `map` to create a list in which every element is the double of those in the original list.

```
List<Integer> lst_doubled = lst.map(Double.function());
System.out.println("Doubled list = " + lst_doubled);
```

with output

```
Doubled list = 2  4  6  8
```

Similarly, we can add an integer constant to every element of a list. Here is a class whose instances are functions that add a given constant (supplied when the instance is created) to an integer.

```
public class AddConstant implements MappingFunction<Integer> {
  private Integer constant;

  private AddConstant (Integer c) { constant = c; }

  public static AddConstant function (Integer c) { return new AddConstant(c); }

  public Integer apply (Integer a) { return a + constant; }
}
```

We can construct a function that adds 3 to an integer with `AddConstant.function(3)`:

```
List<Integer> lst_add3 = lst.map(AddConstant.function(3));
System.out.println("Added 3 to list = " + lst_add3);
```

with output

```
Added 3 to list = 4  5  6  7
```

Programming with something like `MappingFunction<A>` is called *higher-order programming*, where you pass functions around as arguments to other functions, and return functions from functions — generally, treating functions like any other values.

To see the flexibility of such a way to program, consider the problem of applying two transformations in sequence to a list. For instance, suppose you want to double the elements of a list, and then add 1 to each. One way to do that is to simply call `map()` twice, with a different mapping function:

```
List<Integer> lst_dbladd1 =
    lst.map(Double.function()).map(AddConstant.function(1));
System.out.println("Double & add 1 to list = " + lst_dbladd1);
```

with output

```
Double & add 1 to list = 3  5  7  9
```

But there is another way to do this. We can simply create a new transformation that does both the doubling and the adding of 1 to each element, all in one shot. We could write that transformation from scratch, but it's easier to just define a transformation that takes two transformations as inputs, and *composes* them together:

```
public class Compose<A> implements MappingFunction<A> {
  private MappingFunction<A> f1;
  private MappingFunction<A> f2;

  private Compose (MappingFunction<A> af1,
                   MappingFunction<A> af2) {
    f1 = af1;
    f2 = af2;
  }

  public static <B> Compose<B> function (MappingFunction<B> af1,
                                         MappingFunction<B> af2) {
    return new Compose<B>(af1,af2);
  }

  public A apply (A a) {
    return f1.apply(f2.apply(a));
  }
}
```

We can now come up with an alternate way of doubling and adding 1 to the elements of a list (say) as follows:

```
List<Integer> lst_dbladd1alt =
    lst.map(Compose.function(AddConstant.function(1),Double.function()));
System.out.println("Double & add 1 to list (alternate) = " + lst_dbladd1alt);
```

with output

```
Double & add 1 to list (alternate) = 3  5  7  9
```

Of course, we can now compose any mapping functions, such as:

```
List<Integer> lst_dbldbl =
    lst.map(Compose.function(Double.function(),Double.function()));
System.out.println("Double-double list = " + lst_dbldbl);
```

with output:

```
Double-double list = 4  8  12  16
```

The advantages of composing functions like that include the fact that (1) many interesting functions can be obtained by composition of simpler functions, and (2) it requires only a single traversal of the list, as opposed to two or more traversals. This can make a big difference if lists are large.

Mapping functions are restricted in the above interface to transform a value to a value of the same type. It is easy to generalize `map` to return a list of a different type, by using a more general form of `MappingFunction` parameterized over the argument type to `apply()` and the (different) return type of `apply()`. It's a good exercise to write that generalized form of `map()`. Instead of doing that here, I generalize `map` even more below.

14.2 The Reduce Design Pattern

The Map design pattern transforms a structure into a similar structure, by transforming the elments one by one via a mapping function. The Reduce design patterns generalizes this by allowing you to "reduce" a structure into some other kind of data. The idea, for a list `cons(a,L)`, is to recursively reduce the rest `L` of the list into some value V, and then use the current element of the list and the reduced value V to give the reduced result for `cons(a,L)` as a function of the current element and the reduced rest `V`.

Roughly, we want an operation `<B> B reduce (??)` on lists, for some appropriate `??`, with the following specification:

```
empty().reduce(??) = ?? for empty lists
cons(a,L).reduce(??) = ?? for a and L.reduce(??)
```

The question is now, what do we take as `??`, which must tell us how to reduce empty lists, and how to reduce cons lists. We follow the same idea as for map, and define the notion of a list reducing function:

```
public interface ListReducingFunction<A,B> {
  public B applyEmpty ();
  public B applyCons (A elt, B rest);
}
```

Note that a list reducing function is really a pair of functions, one telling us how to reduce an empty list, and one telling how to reduce a list made up of a cons cell. Note also that a list reducing function is parameterized over both a type `A` for the content of the list that we are reducing, and a type `B` for the result of the reduction.

With such a class, we can add and specify the `reduce` operation for polymorphic lists formally:

```
ACCESSORS:    ...
              <B> B reduce (ListReducingFunction<A,B> rf)

SPECIFICATION:    ...
                  empty().reduce(rf) = rf.applyEmpty()
                  cons(a,L).reduce(rf) = rf.applyCons(a,L.reduce(rf))
```

which, when we apply the design pattern to derive an implementation from an ADT specification, give us the following methods added to the list implementation we gave above:

```
public abstract class List<A> {
  ...
  public abstract <B> B reduce (ListReducingFunction<A,B> rf);
}


class EmptyList<A> extends List<A> {
  ...
  public <B> B reduce (ListReducingFunction<A,B> rf) {
     return rf.applyEmpty();
  }
}
```

```
class ConsList<A> extends List<A> {
  ...
  public <B> B reduce (ListReducingFunction<A,B> rf) {
     B r = restElements.reduce(rf);
     return rf.applyCons(firstElement,r);
  }
}
```

Once again, things may be clearer when we look at examples. Recall the sample list `lst` we defined above:

```
List<Integer> lst =
  List.cons(1,List.cons(2,List.cons(3,List.cons(4,List.<Integer>empty()))));
```

Here is a reducing function that reduces a list to an integer representing its length.

```
public class Length<A> implements ListReducingFunction<A,Integer> {
  private Length () {}

  public static <B> Length<B> function () { return new Length<B>(); }

  public Integer applyEmpty () { return 0; }

  public Integer applyCons (A a, Integer rest) { return rest + 1; }
}
```

We see how the reduction will work. An empty list will reduce to the value 0 — and indeed, an empty list has length 0. A list with current element a and where the rest of the list has been reduced to v (that is, where the rest of the list has length v) reduces to $1 + v$.

Thues, we can write:

```
Integer len = lst.reduce(Length.<Integer>function());
System.out.println("Length = " + len);
```

with output

```
Length = 4
```

A slight modification gives us a method to compute the sum of all the elements in the list, by reducing an integer list to an integer representing the sum of the elements.

```
public class Sum implements ListReducingFunction<Integer,Integer> {
  private Sum () {}

  public static Sum function () { return new Sum(); }

  public Integer applyEmpty () { return 0; }

  public Integer applyCons (Integer a, Integer rest) { return a + rest; }
}
```

Here again, an empty list reduces to 0 — the sum of the elements of an empty list, while a list with current element a and whose rest of the list sums to v reduces to $a + v$, which is indeed the sum of the list. Thus, for example:

```
Integer sum = four.reduce(Sum.function());
System.out.println("Sum = " + sum);
```

with output

```
Sum = 10
```

Let's look at a slightly more interest example of reduction, where we want to extract the maximum element of an integer list. We cannot just reduce a list to an integer, because the maximum element may be undefined — the empty list, for example, has no maximum element. To get around that problem, we will have our reduction reduce a list to either an integer or a value indicating that there is no maximum. The cleanest way to do that is to use an ADT for "optional" value, where an optional value is either `none()` or `some(a)`. Operations to check if a value is defined, and to extract the value, are provided. Here is the formal ADT definition.

```
CREATORS:    <A> Option<A> none ()
             <A> Option<A> some (A)

ACCESSORS:   boolean isNone ()
             A valOf ()
             String toString ()

SPECIFICATION:  none().isNone() = true
                some(v).isNone() = false
                some(v).valOf() = v
                none().toString() = "None"
                some(v).toString() = "Some("+v.toString()+")"
```

The implementation is left as an exercise – actually, this is Question 1 of Homework 4. With such an ADT for optional values, we can write a list reducing function to extract the maximum element of a list.

```
public class Max implements ListReducingFunction<Integer,Option<Integer>> {
  private Max () {}

  public static Max function () { return new Max(); }

  public Option<Integer> applyEmpty () { return Option.none(); }

  public Option<Integer> applyCons (Integer a, Option<Integer> rest) {
    if (rest.isNone())
      return Option.some(a);
    if (a <= rest.valOf())
      return rest;
    return Option.some(a);
  }
}
```

An empty list does not have a maximum element, so it reduces to `none()`. A list `cons(a,L)` where the list `L` already has had its maximum extracted (if it exists) as value `v` — which is an optional value — reduces to the maximum of `a` and `v`. For example:

```
Option<Integer> max = lst.reduce(Max.function());
System.out.println("Max = " + max);
```

with output

```
Max = Some(4)
```

More interestingly still, we can use a reduction to compute the reverse of a list. Here, the reducing function returns a list, the list obtained by reversing the current list.

```
public class Reverse<A> implements ListReducingFunction<A,List<A>> {
  private Reverse () {}

  public static <B> Reverse<B> function () { return new Reverse<B>(); }

  public List<A> applyEmpty ()  { return List.empty(); }

  public List<A> applyCons (A a, List<A> rest) {
```

```
    List<A> e = List.empty();
    return rest.append(List.cons(a,e));
  }
}
```

An empty list reduces to the empty list. A list `cons(a,L)`, where `L` has already been reduced (i.e., reversed) into `v`, can be reversed by simply appending `a` at the end of `v`. And indeed, we can check:

```
List<Integer> rev = lst.reduce(Reverse.<Integer>function());
System.out.println("Reverse = " + rev);
```

with output

```
Reverse = 4  3  2  1
```

The key thing is that in one fell swoop, we have obtained several useful functions that transforms lists into other form of data, without having to rewrite the code to do the actual walk down the list. All we had to supply is functions to locally reduce the list assuming that the rest of the list has already been reduced.

As the following exercises should make clear, reduction is in fact quite powerful, and generalizes much of what we've seen until now.

Exercise: *Show that you can implement* `map(mf)` *as* `l.reduce(rf)`*, for a suitably defined* `rf`*. For bonus brownie points, write a class that automatically transform any mapping function* `mf` *to a suitable* `rf` *to do this.*

Exercise: *Show that you can implement* `funcIterator()` *for lists as* `reduce(rf)` *for a suitably defined* `rf`*.*

We have been concentrating on lists, but there is nothing really specific to lists here, aside from the details of the `ListReducingFunction` interface. Any ADT, especially recursive ADTs, can benefit from mapping and reducing.

Exercise: *Consider the following polymorphic tree ADT, with the following signature and specification:*

```
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 ()

SPECIFICATION:
 empty().isEmpty() = true
 node(v,l,r).isEmpty() = false
 node(v,l,r).root() = v
 node(v,l,r).left() = l
 node(v,l,r).right() = r
 empty().toString() = "-"
 node(v,l,r).toString() = v + "[" + l.toString() + "," + r.toString() + "]"
```

(1) Implement this ADT using the Specification design pattern .

(2) Specify and implement a `map` *operation on polymorphic trees.*

(3) Specify and implement a `reduce` *operation on polymorphic trees — you will need to define a* `TreeReducingFunction` *interface.*

14.3 Another Example: Expressions

Let's finish up with another example that illustrates the extent to which many operations are just reductions for a suitably-defined reducing function.

Consider a simple ADT for writing down arithmetic expressions:

```
CREATORS:     Exp lit (int)
              Exp plus (Exp, Exp)
              Exp times (Exp, Exp)
              Exp neg (Exp)

ACCESSORS:    <B> B reduce (ExpReducingFunction<B>)
              String toString ()

SPECIFICATION:
 lit(i).reduce(rf) = rf.applyLit(i)
 plus(e1,e2).reduce(rf) = rf.applyPlus(e1.reduce(rf),e2.reduce(rf))
 times(e1,e2).reduce(rf) = rf.applyTimes(e1.reduce(rf),e2.reduce(rf))
 neg(e).reduce(rf) = rf.applyNeg(e.reduce(rf))
 lit(i).toString() = i
 plus(e1,e2).toString() = "(" + e1.toString() + " + " + e2.toString() + ")"
```

```
  times(e1,e2).toString() = "(" + e1.toString() + " * " + e2.toString() + ")"
  neg(e).toString() = "-" + e.toString()
```

where `ExpReducingFunction` is the following interface:

```
public interface ExpReducingFunction<B> {
  public B applyLit (int i);
  public B applyPlus (B left, B right);
  public B applyTimes (B left, B right);
  public B applyNeg (B val);
}
```

Here is the implementation using the usual design pattern:

```
public abstract class Exp {

  public static Exp lit (int i) {
    return new ExpLit(i);
  }

  public static Exp plus (Exp e1, Exp e2) {
    return new ExpPlus(e1,e2);
  }

  public static Exp times (Exp e1, Exp e2) {
    return new ExpTimes(e1,e2);
  }

  public static Exp neg (Exp e) {
    return new ExpNeg(e);
  }

  public abstract <B> B reduce (ExpReducingFunction<B> rf);

  public abstract String toString ();
}


class ExpLit extends Exp {

  private int value;
```

```
  public ExpLit (int v) {
    value = v;
  }

  public <B> B reduce (ExpReducingFunction<B> rf) {
    return rf.applyLit(value);
  }

  public String toString () {
    return "" + value;
  }
}


class ExpPlus extends Exp {

  private Exp left;
  private Exp right;

  public ExpPlus (Exp l, Exp r) {
    left = l;
    right = r;
  }

  public <B> B reduce (ExpReducingFunction<B> rf) {
    B lred = left.reduce(rf);
    B rred = right.reduce(rf);
    return rf.applyPlus(lred,rred);
  }

  public String toString () {
    return "("+left.toString()+" + "+right.toString()+")";
  }
}


class ExpTimes extends Exp {

  private Exp left;
  private Exp right;

  public ExpTimes (Exp l, Exp r) {
```

```
    left = l;
    right = r;
  }

  public <B> B reduce (ExpReducingFunction<B> rf) {
    B lred = left.reduce(rf);
    B rred = right.reduce(rf);
    return rf.applyTimes(lred,rred);
  }

  public String toString () {
    return "("+left.toString()+" * "+right.toString()+")";
  }
}


class ExpNeg extends Exp {

  private Exp value;

  public ExpNeg (Exp v) {
    value = v;
  }

  public <B> B reduce (ExpReducingFunction<B> rf) {
    B vred = value.reduce(rf);
    return rf.applyNeg(vred);
  }

  public String toString () {
    return ("-"+value.toString());
  }
}
```

For example, here is a sample expression that we can construct, and print:

```
  Exp sample = Exp.times(Exp.plus(Exp.lit(5),
                                  Exp.lit(3)),
                         Exp.plus(Exp.lit(10),
                                  Exp.neg(Exp.plus(Exp.lit(22),
                                                   Exp.lit(33)))));
  System.out.println("Expression = " + sample);
```

with output

```
Expression = ((5 + 3) * (10 +  -(22 + 33)))
```

I claim interesting operations on expressions are simply reductions. First in line, evaluating an expression down to an integer value. Here is an expression reducing function that does evaluation.

```
public class Eval implements ExpReducingFunction<Integer> {

  private Eval () {}

  public static Eval function () { return new Eval(); }

  public Integer applyLit (int i) { return i; }

  public Integer applyPlus (Integer left, Integer right) { return left + right; }

  public Integer applyTimes (Integer left, Integer right) { return left * right; }

  public Integer applyNeg (Integer val) { return (-val); }
}
```

Intuitively, a `lit(i)` expression reduces simply to integer `i`, while a `plus(e1,e2)` expression, once `e1` and `e2` have themselves been reduced (i.e., evaluated) to v_1 and v_2, reduces naturally to $v_1 + v_2$; similarly for `times(e1,e2)` and `neg(e)`. And indeed, we can check this.

```
Integer eval = sample.reduce(Eval.function());
System.out.println("Eval = " + eval);
```

with output

```
Eval = -360
```

A more interesting reduction is to compile an expression down to so-called Reverse Polish Notation, which is an old notation for writing and evaluating expressions without using parentheses, using a stack. The idea is that a "stack code" is a sequence of integers and operations symbols such as $+$, $*$, and $-$, and can be evaluated using a stack as follows:

(1) walk down the stack code;

(2) when you encounter an integer, push it on the stack;

(3) when you encounter +, pop the top two elements of the stack, add them together, and push the result on the stack;

(4) when you encounter $*$, pop the top two elements of the stack, multiply them together, and push the result on the stack;

(5) when you encounter $-$, pop the top element of the stack, negate it, and push the result on the stack.

For example, the stack code `1 2 +` pushes `1` on the stack, then `2` on the stack, then pops `1` and `2` off the stack, adding them together to push the result `3` on the stack. The result of the stack code can be read off the top of the stack. Similarly, `1 2 + 3 4 + *` evaluates to `21`. (Check it!)

I claim that generating the stack code corresponding to a given expression (that is, giving some stack code that evaluates to the same result as a given expression) is just a reduction of the expression, accomplished by the following expression reducing function. We use a list of strings to represent a stack code.

```
public class StackCode implements ExpReducingFunction<List<String>> {

  private StackCode () {}

  public static StackCode function () { return new StackCode(); }

  public List<String> applyLit (int i) {
    List<String> e = List.empty();
    return List.cons("" + i,e);
  }

  public List<String> applyPlus (List<String> left, List<String> right) {
    List<String> e = List.empty();
    return left.append(right.append(List.cons("+",e)));
  }

  public List<String> applyTimes (List<String> left, List<String> right) {
    List<String> e = List.empty();
    return left.append(right.append(List.cons("*",e)));
  }

  public List<String> applyNeg (List<String> val) {
    List<String> e = List.empty();
    return val.append(List.cons("-",e));
  }
```

```
}
```

Intuitively, reducing a literal simply gives the stack code for pushing that literal on the stack; reducing an expression `plus(e1,e2)`, where `e1` and `e2` have been reduced (i.e., translated into stack code) s_1 and s_2, yields the stack code to first perform s_1 then perform s_2 then applying the + operation, which is simply the concatenation of the stack code s_1 and the stack code s_2 and the operation +; similarly for `times(e1,e2)` and `neg(e)`. For example,

```
List<String> stackcode = sample.reduce(StackCode.function());
System.out.println("Stack code = " + stackcode);
```

with output

```
Stack code = 5  3  +  10  22  33  +  -  +  *
```

We can check that `5 3 + 10 22 33 + - + *` indeed is stack code that evaluates to `-360`, as desired.