

Lecture 14: Design Patterns for Traversal

October 8, 2002

In this lecture, we consider design patterns intended for traversing collections of objects.

14.1 Iterator

Suppose you are designing the interface for `Set`, a data type that represents a set of objects. We've already seen a number of data structures that you might use to implement a set: an array, a linked list, a hash table, an ordered tree (q.v. the Java library class `TreeSet`). The question is, what interface should we provide to clients of our set that enables them to iterate over the members of the set, regardless of how the set is actually stored? In pseudocode, the client wants to do this:

```
for each element in set
    do action
```

There are a number of ways we might provide this functionality to clients of our `Set` class. We might provide a method that gets an object at a given index:

```
// try #1
interface Set {
    Object get (int i);
    // returns: element at index i in some ordering of
    //           the set (fixed until the set is modified)
    // throws: IndexOutOfBoundsException if i<0 or i>=size()
    ...
}
```

Clients could then iterate over all the indices from 0 up to the size of the set, getting each element. This interface would be ideal if the set is stored in an array; implementation would be trivial and fast. For a linked list, however, getting good performance would be far from trivial, since `get(i)` would have to count `i` elements down the list. Iterating over all the elements in the list would take quadratic time. A clever optimization can bring this down to linear time by caching the index and list entry of the last element accessed. But it doesn't solve the problem for the hash table representation of a set, or the tree representation. Worse, this interface gives the client the freedom to jump around, accessing arbitrary indices, which is more than the client needs just to iterate over the collection. As implementors, we're letting ourselves in for headaches if we promise too much to our clients.

Another interface might return the objects in the collection as an array:

```
// try #2
interface Set {
    Object[] getAll ();
    // returns: array containing all objects in the set,
    //           whose length is the size of the set
    ...
}
```

Clients would then iterate over the objects in the returned array. This approach is wasteful for large collections, since it has to allocate a new array large enough to hold the collection and copy the collection into it.

The iterator pattern solves this problem. An iterator is an object that yields the elements of a collection one by one, while hiding the details of accessing the collection's representation. In our example, the Set interface would have a method that returns an iterator:

```
interface Set {
    Iterator iterator ();
    // returns: iterator that yields all elements in set
    ...
}
```

The iterator obeys the Iterator interface:

```
interface Iterator {
    boolean hasNext ();
    // returns: true if there are more elements to yield

    Object next ();
    // modifies: this
    // returns: next element in iteration
    // throws: NoSuchElementException if no more elements
    // effects: modifies the iterator to record the yield
}
```

Typically, an iterator is used in a loop guarded by a call to hasNext:

```
Iterator g = set.iterator ();
while (g.hasNext ()) {
    Object x = g.next ();
    ... // use x
}
```

In this context, next can never throw a NoSuchElementException, because the check of hasNext prevents it. Thus there is no need to catch the exception. Since this conventional idiom avoids the exception, NoSuchElementException is declared as an unchecked exception.

Note that an iterator is a mutable object, which is consumed as you use it. Once hasNext returns false, the iterator is used up and should be discarded. To iterate through the collection again, you have to call iterator again to create a fresh one.

14.1.1 Implementing an Iterator

Let's look at the implementation of an iterator. (By the way, the Liskov text calls the class that implements Iterator a *generator*, reserving the term *iterator* for the method called `iterator`. Most programmers use *iterator* to refer to the class.) We'll use the `LinkedList` class that has been discussed in previous lectures:

```
class LinkedList {
    Entry header;
    ...
}

class Entry {
    Object element;
    Entry next;
    Entry prev;
    ...
}
```

Recall that the representation of `LinkedList` is a circular, doubly-linked list of `Entry` objects, in which every `Entry` except the header stores one of the elements in the collection.

We first add an iterator method that allows clients can obtain an iterator:

```
class LinkedList {
    ...
    Iterator iterator () {
        return new LinkedListIterator (this);
    }
}
```

The method creates an instance of `LinkedListIterator`, passing it a reference to the list. `LinkedListIterator` implements the `Iterator` interface:

```
class LinkedListIterator implements Iterator {
    LinkedList list; // list
    Entry curr; // entry before next element to yield

    LinkedListIterator (LinkedList l) {
        list = l;
        curr = l.header;
    }

    boolean hasNext () {
        return curr.next != list.header;
    }

    Object next () {
```

```

        if (!hasNext ())
            throw new NoSuchElementException ();
        curr = curr.next;
        return curr.element;
    }
}

```

This iterator works by advancing `curr` down the list, starting from the header. When the next element is requested from the iterator, it first advances `curr`, and then returns the element found in the new `curr`'s entry. After `curr` reaches the last entry in the list, `hasNext` returns false and the iterator advances no farther.

14.1.2 Rep Invariants and Abstraction Functions for Iterators

The rep invariant for `LinkedListIterator` is:

```

list != null &&
curr == list.header.*next

```

(Recall that `.*next` means that you follow the `next` field zero or more times.) This rep invariant guarantees that `curr` always points to an entry somewhere in the list.

The abstraction function for an iterator maps its rep into a sequence of objects — the objects remaining to be yielded by the iterator. Thus a fresh iterator's abstract value is the sequence containing all the objects in the collection (in the order that the iterator will produce them), while an exhausted iterator corresponds to the empty sequence. The abstraction function for `LinkedListIterator` is:

```

AF(c) = [x1, ..., xn]
where curr.n+1next = list.header
      && xi = curr.inext.value

```

(Here, we're using the notation `curr.inext` to denote the object you get when you start with `curr` and follow the `next` field i times.)

From the abstraction function, you can tell that the iterator is exhausted ($n = 0$) when `curr.next = list.header`; otherwise, the next element to be yielded by the iterator (x_1) is `curr.next.value`.

Iterator implementations are notoriously susceptible to off-by-one errors, since it can be hard to remember whether the iterator's state describes the *last* element yielded or the *next* one. A carefully thought-out abstraction function makes it easier to write `hasNext` and `next` consistently.

14.1.3 Concurrent Modification

A look at the rep invariant points out another problem: what happens if you remove an element from the collection while iterating over it? Consider the following code, whose intent is to remove some of the elements from the set:

```

Iterator g = set.iterator ();
while (g.hasNext ()) {
    Object o = g.next ();
    if (should remove o)
        list.remove (o);
}

```

This code breaks the iterator, because it removes the entry referred to by `curr` from the list. In general, it is unsafe to modify a collection while an iterator is using it. The collection's iterator method should document this as a requirement on the client:

```

interface Set {
    Iterator iterator ();
    // returns: iterator that yields all elements in list
    // requires: can't modify list while using iterator
    ...
}

```

However, since it's so common to iterate through a collection in order to remove elements from it, Java's `Iterator` interface includes an optional `remove` method that can be used to modify the underlying collection safely:

```

interface Iterator {
    ...
    void remove ();
    // modifies: collection associated with this iterator
    // effects: removes last element yielded by next()
    // throws: UnsupportedOperationException if not supported
    // throws: IllegalStateException if next() never called
           or not called since last remove()
}

```

Our `LinkedListIterator` implementation would implement `remove` by first adjusting its own `rep`, then calling `LinkedList.remove` to actually remove the element.

Note that calling `remove` through an iterator does not eliminate the problem of concurrent modification. If you have two iterators active over the same collection — whether in nested loops or concurrent threads — you cannot call `remove` on either iterator, because the change to the collection would break the other iterator.

Some iterators in the Java library throw a `ConcurrentModificationException` when they detect that you have changed the underlying collection between calls to the iterator. You can't count on this exception, but it can help your debugging. An easy way to add concurrent-modification checking to your own iterators is to put a version number on the collection, which is incremented whenever the collection is modified. Then the iterator can easily test whether the collection changed under its feet.

14.1.4 Rep Exposure by an Iterator

Adding `remove` to the `Iterator` interface has a subtle consequence: returning an iterator can expose your rep to modification by clients outside the class. Consider this class, which stores a list of integers and keeps track of their sum:

```
class SumList {
    List elts;
    int sum;

    // Rep invariant:
    // sum = sum of all x in elts

    void add (int x) {
        elts.add (new Integer (x));
        sum += x;
    }

    void remove (int x) {
        if (elts.remove (new Integer (x)))
            sum -= x;
    }

    Iterator iterator () {
        return elts.iterator ();
    }
}
```

By returning an iterator created directly from `elts`, this class exposes its rep: the client may (reasonably) try to call the iterator's `remove` method to remove integers from the list, but doing so would bypass the computation of `sum`, breaking the rep invariant.

One solution would copy `elts` into a new collection and return an iterator over the copy. Another solution would wrap `elts` with a wrapper that makes it immutable, and return an iterator over the immutable wrapper. (See `Collections.unmodifiableList`, a factory method that can wrap an arbitrary `List` to make it immutable.)

But both of these approaches are cop-outs, because they forbid a client from the (very sensible) possibility of using the iterator to remove from the list. The best solution is a custom iterator that delegates to the `List`'s iterator and handles `remove` correctly:

```
class SumList {
    List elts;
    int sum;
    ...
    Iterator iterator () {
        return SumListIterator (this);
    }
}
```

```

class SumListIterator {
    SumList sumlist;
    Iterator g; // List iterator
    Integer last; // last element yielded

    SumListIterator (SumList s) {
        sumlist = s;
        g = s.elts.iterator ();
    }
    boolean hasNext () {
        return g.hasNext ();
    }
    Object next () {
        last = (Integer) g.next ();
        return last;
    }
    void remove () {
        g.remove ();
        sumlist.sum -= last.intValue ();
    }
}

```

14.2 Traversing Hierarchies

Iterators only provide for traversal through a *linear* collection. What about hierarchical collections? Object-oriented software systems are full of hierarchies:

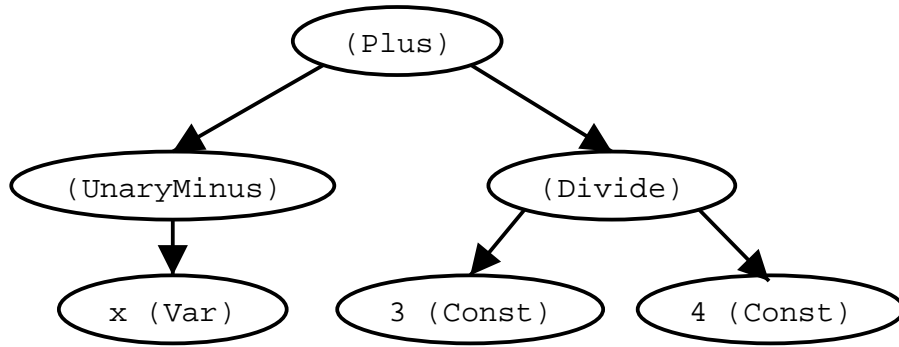
- a user interface is a hierarchical collection of UI objects: windows, panes, buttons, textboxes;
- a filesystem is a hierarchy of directories and files;
- an email client contains a hierarchy of mail accounts, folders, messages, and attachments.

Before we look at how to traverse these kinds of hierarchies, we should first consider how the hierarchy might be represented. This is the goal of the *composite pattern*.

14.2.1 Composite

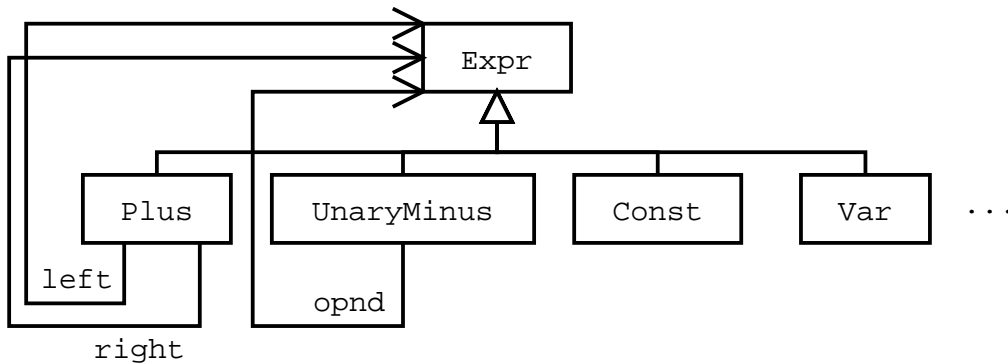
The composite pattern permits a client to manipulate either an atomic unit or a collection of units in exactly the same way. The client need not create special code for the case of a higher-level object with structure as opposed to a basic object. The same operations work on both, because both extend the same type.

Consider the problem of representing expressions in a programming language. Syntactically, the expression $-x + 3/4$ can be represented by a hierarchy of objects called an *abstract syntax tree*:



The nodes at the bottom of the tree, instances of `Const` and `Var`, are the atomic units, or *leaves*. The internal nodes of the tree, corresponding to operators like `Plus` and `Divide`, are *composites*.

The key idea of the composite pattern is that composites and leaves all share a common interface, which in this case we'll call `Expr`. Here's part of the object model showing the relevant classes:



The shape of this object model tells you it's a composite pattern: a base class (`Expr`) with several subclasses, some of which are terminal (the leaves `Const` and `Var`), and some of which have references to subexpressions (the composites `Plus` and `UnaryMinus`).

The code for these classes might look like this, if we only pay attention to the fields and ignore the methods:

```
interface Expr {
    ...
}

class Plus implements Expr {
    Expr left;
    Expr right;
    ...
}

class UnaryMinus implements Expr {
    Expr opnd;
    ...
}
```

```

class Const implements Expr {
    double value;
    ...
}

class Var implements Expr {
    String name;
    ...
}

```

A complete representation would need other classes as well, to represent multiply, divide, subtract, etc.

14.2.2 Interpreter

Now we're ready to talk about patterns for traversing composite structures, using `Expr` as an example. There are many operations we may want to perform on an expression. Suppose we want to evaluate the expression, given some execution context that assigns values to variables. Evaluation might be provided as a method of `Expr`:

```

interface Expr {
    double eval (Context context);
    ...
}

```

Each type of expression then implements `eval` in the appropriate manner. In particular, composites like `Plus` have to recursively evaluate their subexpressions:

```

class Plus implements Expr {
    Expr left;
    Expr right;

    double eval (Context context) {
        return left.eval (context) + right.eval (context);
    }
    ...
}

class Const implements Expr {
    double value;

    double eval (Context context) {
        return value;
    }
    ...
}

```

```

class Var implements Expr {
    String name;

    double eval (Context context) {
        return context.getVarValue (name);
    }
    ...
}

```

This technique is called the *interpreter pattern*. An operation that needs to traverse a composite hierarchy is declared as a method on the composite base class, in this case `Expr`. Composite nodes recursively call the operation on their parts, and the leaves terminate the recursion. In the interpreter pattern, both the operation and the means of traversal are bound up in the composite classes themselves.

14.2.3 Procedural Traversal

A number of operations might be reasonably implemented as interpreter methods on `Expr`: pretty-printing, type-checking, and optimization come to mind. But a problem arises when a client wants to define a new operation. For example, suppose I want to know all the variable names that are used by an expression, perhaps so I can introduce a new variable and avoid conflicts with existing names. With the interpreter pattern, I would have to add a new method to `Expr` for my new operation. Then I'd have to implement the new method in all the subclasses of the composite. I'd need to change many classes to implement just one new operation. Furthermore, I'd need access to the source code of the expression classes, which isn't always possible.

A first step towards fixing this problem is to represent the new operation as a class itself. (This is what the Liskov text calls the *procedural approach* to hierarchy traversal, although here I'm using a class rather than a collection of static procedures.) The class has a separate method for each type of expression. Methods corresponding to composites take care of traversing the composite's children. Here's the code:

```

class FindVariables {
    Set vars = new HashSet ();

    void forExpr (Expr e) {
        ... // see below for this method's body
    }

    void forPlus (Plus e) {
        forExpr (e.left ());
        forExpr (e.right ());
    }

    void forConst (Const e) { }
    void forVar (Var e) {
        vars.add (e.name ());
    }

    ...
}

```

This class traverses the expression tree, and every time a `Var` node is encountered, its variable name is added to a set. For convenience of other clients, it's probably best to hide this operation class behind a simple method interface that instantiates it and invokes it correctly:

```
static Set variablesUsed (Expr e) {
    FindVariables op = new FindVariables ();
    op.forExpr (e);
    return op.vars;
}
```

This works well enough — at least we didn't need to change the expression classes — but it has one ugly aspect that was omitted from the code above. The method for a composite like `forPlus` doesn't know what method it should call for its subexpressions, since they could be any type of node. So it calls `forExpr`, which tests for and dispatches on all the possible expression types:

```
void forExpr (Expr e) {
    if (e instanceof Plus) forPlus ((Plus) e);
    else if (e instanceof Const) forConst ((Const) e);
    else if (e instanceof Var) forVar ((Var) e);
    ...
    else assert (false); // don't know e's type
}
```

Maintaining this code is tedious and error-prone. The long list of cascaded `if` tests are likely to run slowly. Furthermore, even though this code is bad enough appearing once, in fact it must occur repeatedly in *every* operation class we define. Systematic repetition in code is usually a sign of a need to redesign, possibly using a pattern.

We already know a Java construct that automatically chooses code to execute based on the type of an object: method dispatch. Method dispatch does the same kind of comparison and selection as the cascaded `if` tests, but does not clutter the code, is likely to be more efficient, and gives us compile-time checking if we forget a case. The *visitor pattern* takes advantage of this.

14.2.4 Visitor

The visitor pattern encodes a traversal over a composite hierarchy. As in the procedural approach, an operation is represented by a class, called a *visitor*, with a method for each type of node in the composite. However, instead of putting type-testing code in the visitor to decide which method needs to be called, that responsibility is shifted to the composite classes. Furthermore, the composite classes will also assume responsibility for the traversal.

Here's the basic scheme:

1. A composite node is asked to *accept* a visitor by a call to its `accept` method:

```
composite.accept (visitor)
```

2. The composite recursively passes the visitor on to its parts:

```
    for each part in composite,  
        part.accept (visitor)
```

3. Depending on its type, the composite calls the appropriate method of the visitor, passing itself as an argument.

```
    visitor.forNodeType (composite)
```

The `accept` and `visit` methods work together in such a way that `composite.accept(visitor)` performs a depth-first traversal of the structure rooted at `composite`. In this case, the traversal is called *postorder* because a node is visited after all its children have been visited. Other possibilities are *preorder*, which visits the node first, and *inorder*, which visits the node between its two children and makes sense only for binary trees.

For our expression example, the code looks like this:

```
interface Visitor {  
    void forPlus (Plus e);  
    void forConst (Const e);  
    void forVar (Var e);  
    ...  
}  
  
interface Expr {  
    void accept (Visitor v);  
    ...  
}  
  
class Plus implements Expr {  
    Expr left;  
    Expr right;  
  
    void accept (Visitor v) {  
        left.accept (v);  
        right.accept (v);  
        v.forPlus (this);  
    }  
    ...  
}  
  
class Const implements Expr {  
    String name;  
  
    void accept (Visitor v) {  
        v.forConst (this);  
    }  
    ...  
}
```

```

class Var implements Expr {
    String name;

    void accept (Visitor v) {
        v.forVar (this);
    }
    ...
}

```

Using the visitor pattern, the `FindVariables` operation could be implemented much more simply. There is no need for instanceof tests or downcasts, or even any traversal code. In fact, only the `forVar` method has a nontrivial body:

```

class FindVariables implements Visitor {
    Set vars = new HashSet ();

    void forPlus (Plus e) { }
    void forConst (Const e) { }
    void forVar (Var e) { vars.add (e.name ()); }
    ...
}

```

However, the method that uses `FindVariables` must reverse the way it invokes the operation. Instead of passing the expression to the operation, it must pass the operation to the expression:

```

static Set variablesUsed (Expr e) {
    FindVariables op = new FindVariables ();
    e.accept (op);
    return op.vars;
}

```

This validates the decision we made to hide `FindVariables` behind a method interface. Otherwise, changing it to a visitor would have entailed changing every place in the code where it was invoked.

A visitor is very much like an iterator. Each element of the hierarchy is presented to the visitor in turn, with the additional benefit (not provided by the iterator pattern) of a dispatch based on the type of the element. Furthermore, a visitor can accumulate state over the course of its traversal, e.g., the set of variable names encountered.

Unfortunately, the visitor pattern shown above does not give the visitor any control over the traversal. What if we wanted to implement `eval` as a visitor, rather than an interpreter? This would be hard as it stands. To implement `Plus`, for example, the visitor would need the results of evaluating its two subexpressions, but the design above only makes it easy to receive the result of the *last* node that was visited.

The Liskov text proposes one solution to this problem: saving results on a stack and popping them off at the appropriate times. This keeps the visitors and acceptors clean, but it can be hard to see how data flows between visitor calls.

Another solution is to move responsibility for the traversal back into the visitor. We'll call it a `SelfGuidedVisitor`¹ to distinguish it from the more passive kind of visitor, but its methods are superficially the same:

¹This isn't standard terminology, but I haven't seen any good name for this variant on the visitor pattern.

```

interface SelfGuidedVisitor {
    void forPlus (Plus e);
    void forConst (Const e);
    void forVar (Var e);
    ...
}

```

When given a self-guided visitor, the accept methods do nothing but type-dispatching:

```

interface Expr {
    void accept (SelfGuidedVisitor v);
    ...
}

class Plus implements Expr {
    Expr left;
    Expr right;

    void accept (SelfGuidedVisitor v) {
        v.forPlus (this);
    }
    ...
}

class Var implements Expr {
    String name;

    void accept (SelfGuidedVisitor v) {
        v.forVar (this);
    }
    ...
}

```

The self-guided visitor must take over its own traversal, by asking the children of each visited node to accept it. Because it controls when children are visited, the visitor can also keep track of the results returned by those children:

```

class Evaluator implements SelfGuidedVisitor {
    Context context; // maps variables -> values
    double result; // value of last subexpr evaluated

    void forPlus (Plus e) {
        // evaluate left subexpression
        e.left ().accept (this);
        // hang onto its result
        double left = result;
    }
}

```

```
        // evaluate right subexpression
        e.right ().accept (this);
        double right = result;

        // combine the two results
        result = left + right;
    }

    void forConst (Const e) {
        result = e.value ();
    }

    void forVar (Var e) {
        result = context.getVarValue (e.name ());
    }
    ...
}
```

Both variants of the visitor pattern are useful members of your toolbox, and a well-designed composite hierarchy may need to support both kinds.