Kevlin Henney/kevlin@curbralan.com

A tale of three patterns

THE CONTROL-FLOW structures of Java have remained fairly stable since their arrival in C three decades ago. They express the core structured-programming primitives of sequential, selective, and repetitive execution, as well as discontinuous control flow, such as breaking from a loop. Selective execution is much as it used to be, except it has lost some ground to the use of polymorphism. The basic refinement to sequential execution has been its disruption through exceptions and its multiplication through threading, both of which are supported in the heart of the language. My previous two columns[1, 2] explored patterns that captured differences in practice between single-threaded and multi-threaded programming, and exception-free and exception-throwing code.

Iteration constructs have changed little since days of yore, but the subject of iteration has been objectified and pattern-enriched with collections and iterators. To begin at the beginning, there are five features that define any iteration:

- The initialization to start the loop.
- The continuation condition to keep it running (or termination condition to stop it, depending on whether you regard the glass as half-full or half-empty).
- The body of the loop to do the work.
- The advancement of the iteration to go around again. This may be a distinct action, i.e., an increment, or a side effect of something in the body of the loop.
- The finalization to clean up after the loop. This feature is often optional, a no-op.

It is no coincidence that the four mandatory parts of a loop correspond to the anatomy of the for.

The ITERATOR pattern[3] represents what many OO developers think of as *the* design for iteration in an object system, but there is more to iteration abstraction than just ITERATOR. The rest of this column looks at three iteration patterns whose solutions differ in their response to threading, distribution, exceptions, and parameterization. So let's start by looking in detail at something that should be fairly familiar.

Iteration Pattern 1: ITERATOR

Support iteration over an encapsulated collection by placing responsibility for iteration into a separate object.

Kevlin Henney is an independent consultant and trainer based in the UK.

Problem. How can elements of a collection be accessed conveniently, sequentially, and efficiently without exposing its underlying representation or making it less cohesive?

Iteration constructs have changed little since days of yore, but the subject of iteration has been objectified and pattern-enriched with collections and iterators.

If the representation of a collection is accessible, users can access the internal structure of the object directly so they can iterate over it. However, users can undermine its structure, intentionally or otherwise, giving rise to subtle bugs or assumptions about undocumented class behavior. Their code is also strongly coupled to the implementation details of the collection, so any changes to the data structure will inevitably break the client code, and any change of collection type will inevitably cause a rewrite. It is not possible to work reasonably through an interface if the implementation of a collection must be known in order to use it.

It is tempting to use a conventional integer index to look up elements. However, this is an abstraction too far, hiding potentially expensive operations behind an innocent-looking interface. To loop over every element in an array-based sequence with an index is a linear-time operation ($O(N)$ in O-notation), with the generated code visiting each element precisely once in constant time. To do the same for a linked-list implementation is significantly more expensive (a pricey $O(N^2)$) as previously visited links in the list are retraversed each time to find the next indexed element. To get a feel for what this means in practice, an array of 10 elements will require 10 accesses, whereas a list of 10 will require 55 link traversals. But where an array of 100,000 elements requires 100,000 accesses, a list of the same length requires more than 5 billion link traversals. Hiding such a profound difference between a random-access structure and a sequential-access structure behind a weakly specified get interface (no non-functional guarantees or warnings are given) is questionable design.

In sequential-access cases, it can be more effective to make use of a toArray method to get a snapshot of all the elements and iterate over the returned array instead. However, this still requires the user to know the imple-

mentation type of the collection, and it is not without its own overhead.

Iteration can be supported directly in the collection by providing the collection with the notion of a current position. Unfortunately, this restricts a collection to only a single iteration at a time. The methods of the collection cannot use the iteration mechanism for fear they may upset an existing iteration; a collection may be referenced by many objects, each with its own use for the notion of a "current position," and there is therefore a strong possibility of interference between them. There is also a question of cohesion—or rather the lack of it—when a collection's interface and implementation are asked to represent both a collection of items and a currently selected item.

Solution. Place responsibility for iteration into an object separate from the object being iterated over. The fundamental features of iteration must be supported, with the most common division of features being for the collection to support creation and initialization of an iterator object, often using a FACTORY METHOD,[3] and the iterator object itself supporting the traversal features, either as separate methods or as combined methods—COMMAND-QUERY SEPARATION vs. COMBINED METHOD.[1]

From the standard java.util package, the Iterator and Collection interface offer an example of the ITERATOR pattern in action:

```
public interface Iterator
{
  boolean hasNext();
  Object next(); // Combined Method for advance and retrieve
  ...
}
public interface Collection
{
   ...
   Iterator iterator(); // Factory Method
   ...
}
```

The underlying collection data structure is now fully hidden, but without hiding complexity costs from the user. There may be some additional overhead as a result of creating an additional object and adding a managed level of indirection. Whether this is significant depends on the task the iteration is being carried out for and the nature of the application.

The user is now freed from any dependency on implementation details—to the extent that a collection class may be modified without affecting the code, or an alternative collection type may be substituted, assuming an interface-based approach is taken. It is even possible for the collection to be virtual in the sense there is no real collection of objects at all, i.e., objects on an input stream or objects that are calculated on the fly.

There is coupling between the iterator and the collection classes, suggesting the iterator class should be nested within the collection class definition. It may be as a public class in its own right or, more likely, a private inner or static class implementing an interface such as java.util.Iterator. If the iterator is simple, it may be reasonable to define it as an anonymous inner class within the body of the actual creation method.

Multiple traversals can now be supported, but the question of validity arises if the collection is modified while the iteration is carried out. The decision has to be taken as to whether invalidating iterators is reasonable or not for a given design, and then what to do about it.

- For small collections an iterator can take a snapshot of the collection's contents, i.e., such as using toArray and iterate over that. This has the disadvantage of a costly initialization, but the advantage of guaranteeing iteration validity and content, even with multiple threads.
- Another approach to managing iterator validity is to establish an OBSERVER relationship[3] between the iterator and the collection, so that the iterator is notified of any changes in the collection, i.e., either invalidating itself or advancing itself an element if its current element is removed.
- A version stamp can also be used, so the collection has a counter that is only modified when the collection structure is modified. An iterator remembers the version of its creation and always checks to see that the target collection has not changed. This approach is used in the Java Collections API, but has the disadvantage that any change—even one that would not invalidate the current iterator—will cause an exception to be thrown.
- A more tolerant and laissez-faire approach is to only throw an exception when the iterator absolutely cannot perform an operation due to changes. For instance, increasing the size of an array-based collection need not invalidate iterators, but reducing the size below the index of the current iterator would. Removing elements is generally safe from a linked list, and can also be made safe if the current element is removed—either the outgoing references continue to refer to adjacent links to allow iteration to continue, or the outgoing references are nulled, leading to an exception on the next advance.
- The move to the next element, the extraction of the element, and the check that it is actually there can all be combined into a single method. This use of COMBINED METHOD[1] ensures that the traversal aspects of the loop are uninterruptible and can be synchronized by the iterator rather than the user of the iterator. This technique can be combined with any of the strategies above. Note that although java.util.Iterator has a COMBINED METHOD, it does not fulfill the role just described.

The separation of iteration from the interface of the collection means that collection users can customize their own iteration by adapting the raw iterator, i.e., defining a search iterator that skips elements that do not

match some particular condition when it traverses.

Iteration Pattern 2: Enumeration Method

Support encapsulated iteration over a collection by placing responsibility for iteration into a method on the collection.

Problem. Some collection types have representations that do not conveniently support Iterator-based traversal. How can elements of such a collection be sequentially and efficiently accessed without complicating and compromising the implementation of the collection?

For instance, a dictionary implemented as a sorted binary tree can be implemented minimally with each tree node having a link to the child node on the left and on the right, plus the actual data element, and so on recursively down the subtrees. The subtree on the left has values that compare lower than the data element, while those in the subtree on the right compare higher:

```
public class SortedBinaryTree
{
    ...
    private static class Node
    {
        String key, value;
        Node left, right;
    }
    private Node root;
}
```

The simplest approach for visiting each element in order is to recurse through the subtree on the left before dealing with the key and value in the current node, and then recurse through the subtree on the right. However, this data structure will not easily support an iterator. The traversal is broken up, and the context for recursion is lost. One way to allow the iterator to retain context is to add a parent link so the iterator can navigate both up and down a tree:

```
public class SortedBinaryTree
{
    ...
    private static class Node
    {
        String key, value;
        Node left, right, up;
    }
    private Node root;
}
```

However, this complicates the management of the tree and adds a small overhead—sometimes insignificant—to each node. An alternative Iterator implementation does not require a parent link, and captures the depth context itself in a Stack:

```
public class SortedBinaryTree
{
    public class static Iterator implements java.util.Iterator
    {
        ...
        private java.util.Stack nodes; // nodes to revisit
    }
    ...
}
```

The stack places the complexity in the iterator rather than the collection, but can be fragile if the collection's structure changes during iteration.

In each case, there are further issues if something like the standard Iterator interface is used for a dictionary. It provides only for returning a single value, and so a combined data structure to represent the key-value pair must be used. Note that this problem does not arise if Command-Query Separation is used.

More generally, there are times when a collection requires some pre- and post-iteration code to be executed before and after the traversal. The most obvious and common case is synchronization against threaded interruption. Leaving users to do it for themselves is tedious and error prone and, specifically referring to thread synchronization, a synchronized block is problematic because it can give the illusion of safety without any of the safety.[1]

Solution. Bring the iteration inside the collection and encapsulate it in a single method responsible for complete traversal. The task of the loop—what would normally be its body—is passed in as a Command object[3] and is applied to each element in turn (see Listing 1).

The method applies any of the relevant pre- and post-iteration actions itself, performing the loop in between, effectively atomically. Different loop strategies can be supported easily by adding new methods, rather than requiring a whole new object type, i.e., for pre-order traversal. This is particularly useful for recursive data structures, such as Composite objects.[3]

The Enumeration Method[4] is a fundamental iteration pattern that encapsulates the actual control flow of the traversal loop. There are strong similarities with the Template Method design pattern,[3] as well as Visitor[3] and Execute-Around Method.[2] Enumeration Method can be considered an inversion of the Iterator model, both in terms of its flow of control and responsibilities. The performance of an Enumeration Method is directly comparable to that of an Iterator.

Iteration Pattern 3: Batch Method

Group multiple collection accesses together to reduce the cost of multiple individual accesses.

Problem. How can many actions be treated atomically? In particular, how can multiple operations and accesses on a collection be handled efficiently and without interruption? In a distributed system—or any other environment involving access latency, such as database access—iterated simple operations use up bandwidth:

```
for(each key of interest)
{
    ...
    dictionary.put(key, value);
}

for(each key of interest)
{
    value = dictionary.get(key);
    ...
}
```

While ITERATOR allows traversal of a sequence in an abstract and controlled fashion, it does not itself address concurrency and efficiency issues. A COMBINED METHOD addresses the basic concurrency issues, and can fold three operations into one; however, this only reduces the number of accesses, not the number of times a cost-incurring loop is performed.

Solution. Define a single method that performs the action repeatedly. The method is declared to take all the arguments for each execution of the action, i.e., an array or collection, and to return results by similar means. This single method folds the repetition into a data structure rather than a loop, so that looping is performed before or after the method call, in preparation or in follow up. Therefore, the cost of access is reduced to a single access, or a few "chunked" accesses:

```
interface RemoteDictionary ...
{
    Object[] get(Object[] keys);
    void put(Object[] keys, Object[] values);
    ...
}
```

Each method access is now more expensive, but the overall cost has been reduced. Such accesses can also be synchronized as appropriate within the method call. The trade-off in complexity is that significantly more housekeeping is performed to set up and work with the results of the call, and more intermediate data structures are required. The BATCH METHOD pattern is found in many distributed system architectures, such as the CORBA Common Object Services.[5] It can be considered as taking the COMBINED METHOD pattern to its logical extreme in combining many iterations into a single shot.

Conclusion

What drives pattern-based design is not that patterns embody neat fragments of design—although this is often true—but that any given pattern can be considered in tension with one or more other patterns. These other patterns compete as design alternatives, with the trade-offs in their forces and consequences making all the difference, pulling the design one way and another. Sometimes there is a clear winner and the consideration of alternatives does not add anything except time spent mulling over the detail at the water cooler, on the whiteboard, or on the way home. However, at other times having a default case you always apply, without question, can have quite the opposite effect, accidentally brushing significant concerns under the carpet. It is here that having patterns presented together and in opposition captures more of the spirit of the design, even in something as humble and everyday as iteration.

LISTING 1

Iteration over a sorted binary tree using recursive descent in an enumeration method

```
public interface KeyValueTask
{
    void apply(String key, String value);
}

public class SortedBinaryTree
{
    public synchronized void forEachDo(KeyValueTask toDo)
    {
        forEachDo(toDo, root);
    }
    private void forEachDo(KeyValueTask toDo, Node node)
    {
        if(node != null)
        {
            forEachDo(toDo, node.left);
            toDo.apply(node.key, node.value);
            forEachDo(toDo, node.right);
        }
    }
    ...
}

tree.forEachDo(new KeyValueTask() {
    public void apply(String key, String value)
        { System.out.println(key + " -> " + value); }
});
```

References

1. Henney, K. "A Tale of Two Patterns," *Java Report*, Vol. 5, No. 12, December 2000, available at http://www.curbralan.com.
2. Henney, K. "Another Tale of Two Patterns," *Java Report*, Vol. 6, No. 3, March 2001, available at http://www.curbralan.com.
3. Gamma, E., et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
4. Beck, K. *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.
5. Object Management Group (OMG). *CORBAservices: The Common Object Services Specification*, available at http://www.omg.org.