Chapter 24

Iterator

24.1 The Problem

The Java Collection Library contains a long list of different collections each with benefits and liabilities in terms of flexibility, performance versus memory trade-offs, etc. However, we normally want to decouple our client code as much as possible from the implementation details of the data structures. One particular aspect is iteration. Iterating over the elements in a linked list (following the next reference) is radically different from iterating in an array (indexing). This creates a strong coupling between client code and the choice of data structure: if you change your mind and want to use a linked list instead of an array, you have to modify all the iterations.

24.2 A Solution

Again, this problem can be rephrased as a variability problem that can be addressed by the ③-①-② process.

③ *Encapsulate what varies.* Iteration is variable depending on the data structure wanted. Thus iteration itself is the variable behavior to encapsulate.

① *Program to an interface.* Create an interface that contains the central responsibilities of iteration: get the next element; and test if there are any more elements left to iterate.

② *Object composition.* Instead of the client doing the iteration itself, it delegates this to the iterator object.

The protocol between the client and the collection is for the client to request an iterator object. Given this, the client may iterate over all elements. This type of Java code pervades all collection oriented programming.

```
Collection<Item> c = ...;
Item current;
for ( Iterator<Item> i = c.iterator(); i.hasNext(); ) {
  current = i.next();
  [process current]
}
```

24.3 The Iterator Pattern

This is the ITERATOR pattern (design pattern box 24.1, page 41). Its intent is

> *Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.*

The central roles in the iterator is the **Collection** and **Iterator**. Collection is responsible for storing elements while iterator is responsible for the iteration: keep track of how far the iterator has moved, and retrieving the next element, etc.

Iterator is a central design pattern in the Java Collection Framework. It is even supported by the language by the *for-each loop*; if a class implements Iterable (as does all collections) you may write the iteration in the previous section as:

```
for (Item current : c) {
  [process current]
}
```

> ☞ Review the Java Collection Framework in the *java.util* package for its use of iterators.

Developers use iterators without much thought but you may define many interesting iterators that do more interesting things than just enumerating elements from start to end. You may define an iterator that returns elements in random order instead of shuffling the deck of cards. Trees may be iterated breadth-first or depth-first. Return the elements in a collection in reverse order or only every second element.

Many board games are played on a matrix, for instance chess or checkers. Here iterators may list all possible moves that, say, a knight may make from a given square on the board.

```
for ( Iterator<Square> moves =
        board.getKnightPositionIterator(position);
      moves.hasNext(); ) {
  Square s = moves.next();
  [evaluate benefit of moving here]
}
```

Iterators are often implemented as inner classes in Java. The inner class allows the iterator to access the detailed data structure of the collection implementation without exposing it.

The iterator provides several benefits. The compositional design means the collection role and iterator role are smaller and more cohesive abstractions. You can, as argued above, define many different types of traversals but the traversal algorithm, the for loop, always looks the same. Finally, you may have several different iterations pending on the same collection at the same time, as the iteration state is stored in the iterator itself.

One issue to consider if whether the iterator is **robust** or not. The issue arises when a collection is modified while an iterator is traversing it. Non-robust iterators may give incorrect results like returning the same element twice or not at all during the traversal. Robust iterators, however, guaranty to make a proper traversal—like for instance by making a copy of the collection before starting the iteration. Making robust iterators is a complex topic that I will not discuss further.

Iterator's liabilities should be obvious. In contrast to the simple integer index into Java's array construct, iterator is complex with additional interfaces, classes, and complex protocol. The introduction of the for-each loop has lessened this complexity in Java, and similar constructs exist in C#.

24.4 Review Questions

Describe the ITERATOR pattern. What problem does it solve? What is its structure and what is the protocol? What roles and responsibilities are defined? What are the benefits and liabilities?

Describe some non-standard iterations that are easily made by custom iterators.

24.5 Further Exercises

Exercise 24.1. Source code directory:
`exercise/iterator/chess`

Consider a chess game that is played on a 8×8 board on which each square is identified by a Position class encapsulating row and column (or rank and files). Lower, left corner is (1,1), black queen is at (8,4), etc.

To get all the squares that a knight at a given position (r,c) can attack, you can define an iterator that returns all valid positions, like:

Fragment: exercise/iterator/chess/Position.java

```
public static Iterator<Position>
        getKnightPositionIterator(Position p) {
```

1. Implement the knight positions iterator.
2. Implement an iterator that will return all valid positions that a rook may move to (on an empty board).

3. Implement an iterator that will return all valid positions that a bishop may move to (on an empty board).
4. Implement an iterator that will return all valid positions that a queen may move to (on an empty board).

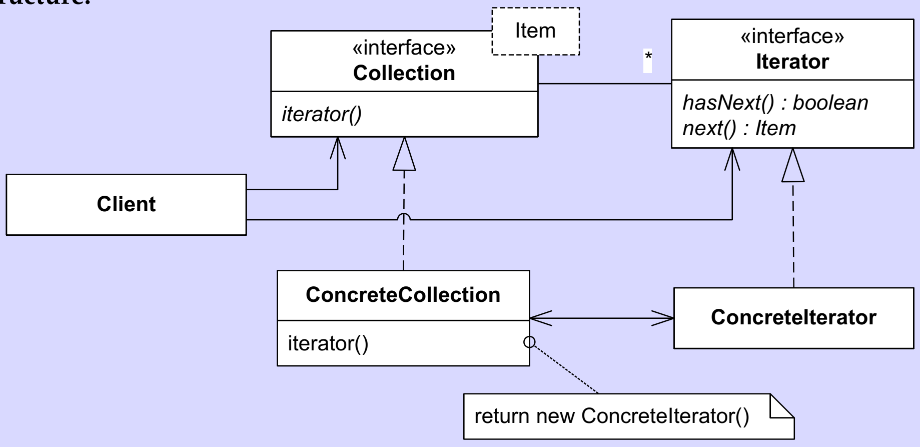
[24.1] Design Pattern: Iterator

Intent Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Problem You want to iterate a collection without worrying about the implementation details of it.

Solution Encapsulate iteration itself into an object whose responsibility it is to allow access to each element in the collection.

Structure:



Roles **Collection** is some data structure that can be iterated. The **Iterator** defines the iteration responsibility, and the collection can upon request return an **ConcreteIterator** that knows how to iterate the specific **ConcreteCollection**.

Cost - Benefit It *decouples iteration from the collection* making client code more resilient to changes to the actual collection used. It *supports variations in the iteration* as the collection can provide a variety of iterators. The *collection interface is slimmer* as it does not itself need to provide iteration methods. You can have *several iterators working on the same collection at the same time* as each iterator holds its own state information. A liability is the *added complexity in protocol* in contrast to traditional iteration.