Design Pattern: Iterator

Intent

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

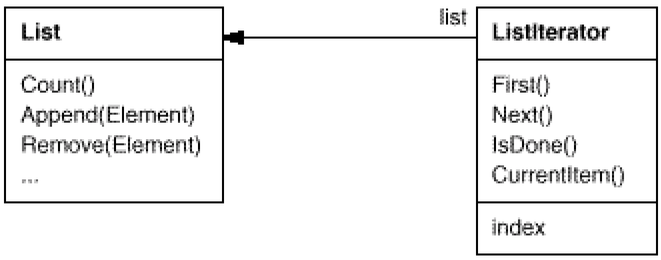
Cursor

Motivation

An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list.

The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

For example, a List class would call for a ListIterator with the following relationship between them:



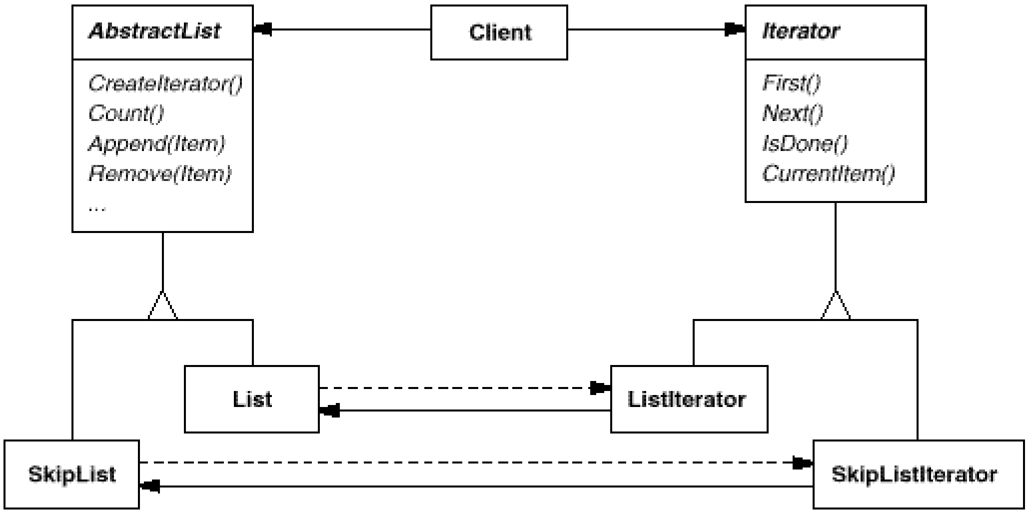
Before you can instantiate ListIterator, you must supply the List to traverse. Once you have the ListIterator instance, you can access the list's elements sequentially. The CurrentItem operation returns the current element in the list, First initializes the current element to the first element, Next advances the current element to the next element, and IsDone tests whether we've advanced beyond the last element—that is, we're finished with the traversal.

Separating the traversal mechanism from the List object lets us define iterators for different traversal policies without enumerating them in the List interface. For example, FilteringListIterator might provide access only to those elements that satisfy specific filtering constraints.

Notice that the iterator and the list are coupled, and the client must know that it is a list that's traversed as opposed to some other aggregate structure. Hence the client commits to a particular aggregate structure. It would be better if we could change the aggregate class without changing client code. We can do this by generalizing the iterator concept to support **polymorphic iteration**.

We define an AbstractList class that provides a common interface for manipulating lists. Similarly, we need an abstract Iterator class that defines a common iteration interface. Then we can define concrete Iterator subclasses for the different list implementations. As a result, the iteration mechanism becomes independent of concrete aggregate classes.

Structure



The remaining problem is how to create the iterator. Since we want to write code that's independent of the concrete List subclasses, we cannot simply instantiate a specific class. Instead, we make the list objects responsible for creating their corresponding iterator. This requires an operation like CreateIterator (or iterator() in Java) through which clients request an iterator object.

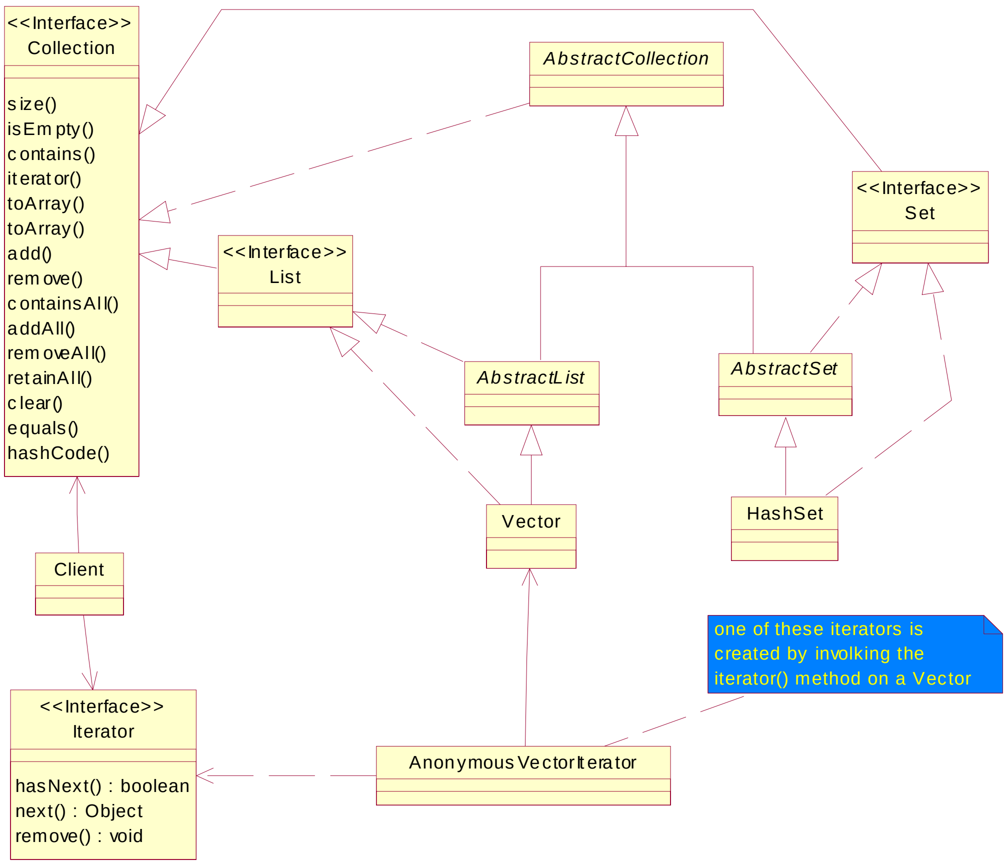
CreateIterator is an example of a factory method (see Factory Method (107)). We use it here to let a client ask a list object for the appropriate iterator. The Factory Method approach give rise to two class hierarchies, one for lists and another for iterators. The CreateIterator factory method "connects" the two hierarchies.

Applicability

Use the Iterator pattern

- to access an aggregate object's contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

In Java (java.util package)



Participants

]

- Iterator : defines an interface for accessing and traversing elements.
- ConcreteIterator : implements the Iterator interface, keeps track of the current position in the traversal of the aggregate.
- Aggregate : defines an interface for creating an Iterator object.
- ConcreteAggregate : implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Collaborations

A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences

The Iterator pattern has three important consequences:

1. It supports variations in the traversal of an aggregate. Complex aggregates may be traversed in many ways. For example, code generation and semantic checking involve traversing parse trees. Code generation may traverse the parse tree inorder or preorder. Iterators make it easy to change the traversal algorithm: Just replace the iterator instance with a different one. You can also define Iterator subclasses to support new traversals.

2. Iterators simplify the Aggregate interface. Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.

3. More than one traversal can be pending on an aggregate. An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.

Implementation

Iterator has many implementation variants and alternatives. Some important ones follow. The trade-offs often depend on the control structures your language provides. Some languages (CLU [LG86], for example and Java) even support this pattern directly.

1. Who controls the iteration? A fundamental issue is deciding which party controls the iteration, the iterator or the client that uses the iterator. When the client controls the iteration, the iterator is called an external iterator (C++ and Java), and when the iterator controls it, the iterator is an internal iterator (Lisp and functional languages). Clients that use an external iterator must advance the traversal and request the next element explicitly from the iterator. In contrast, the client hands an internal iterator an operation to perform, and the iterator applies that operation to every element in the aggregate.

External iterators are more flexible than internal iterators. It's easy to compare two collections for equality with an external iterator, for example, but it's practically impossible with internal iterators. Internal iterators are especially weak in a language like C++ that does not provide anonymous functions, closures, or continuations like Smalltalk and CLOS. But on the other hand, internal iterators are easier to use, because they define the iteration logic for you.

2. Who defines the traversal algorithm? The iterator is not the only place where the traversal algorithm can be defined. The aggregate might define the traversal algorithm and use the iterator to store just the state of the iteration. We call this kind of iterator a **cursor**, since it merely points to the current position in the aggregate. A client will invoke the Next operation on the aggregate with the cursor as an argument, and the Next operation will change the state of the cursor.

If the iterator is responsible for the traversal algorithm, then it's easy to use different iteration algorithms on the same aggregate, and it can also be easier to reuse the same algorithm on different aggregates. On the other hand, the traversal algorithm might need to access the private variables of the aggregate. If so, putting the traversal algorithm in the iterator violates the encapsulation of the aggregate.

3. How robust is the iterator? It can be dangerous to modify an aggregate while you're traversing it. If elements are added or deleted from the aggregate, you might end up accessing an element twice or missing it completely. A simple solution is to copy the aggregate and traverse the copy, but that's too expensive to do in general.

A robust iterator ensures that insertions and removals won't interfere with traversal, and it does it without copying the aggregate. There are many ways to implement robust iterators. Most rely on registering the iterator with the aggregate. On insertion or removal, the aggregate either adjusts the internal state of iterators it has produced, or it maintains information internally to ensure proper traversal.

Kofler provides a good discussion of how robust iterators are implemented in ET++ [Kof93]. Murray discusses the implementation of robust iterators for the USL StandardComponents' List class [Mur93].